UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE INGENIERÍA DEL SOFTWARE E INTELIGENCIA ARTIFICIAL

TESIS DOCTORAL

Arquitectura de pizarras distribuidas para sistemas de Inteligencia Ambiental

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

José María Fernández de Alba López de Pablo

Directores

Rubén Fuentes Fernández
Juan Luis Pavón Mestras

Madrid, 2014

©José María Fernández de Alba López de Pablo, 2014

Arquitectura de pizarras distribuidas para sistemas de Inteligencia Ambiental

TESIS DOCTORAL

José María Fernández de Alba López de Pablo

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Mayo 2014

Documento maquetado con TEXIS v.1.0.

Arquitectura de pizarras distribuidas para sistemas de Inteligencia Ambiental

Memoria que presenta para optar al título de Doctor en Informática

José María Fernández de Alba López de Pablo

Dirigida por los Doctores

Rubén Fuentes Fernández
Juan Luis Pavón Mestras

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Mayo 2014

A Carlos, por aquellas conversaciones de sobremesa en la que concluíamos que todo está mal hecho. Te echo de menos.
Y a Damiano, por ese breve pero intenso empujón de ánimo para acabar todo esto. Eres un crack.

A Lisa, porque sólo ella sabe lo que me ha pasado por la cabeza durante todo este tiempo. Grazie Amore.

Agradecimientos

Forrest, nunca te agradecí haberme salvado la vida.

Teniente Dan Taylor. Forrest Gump.

En primer lugar, debo dar las gracias a mis dos directores, Juan y Rubén, por su sabio consejo e incondicional apoyo, y por su inestimable guía en mi viaje por el mundo académico. Si no fuese por los certeros comentarios de Rubén y por el implacable liderazgo de Juan, seguramente habría acabado cayendo en algún pozo (metafórico o no) por el camino.

Gracias también a mi familia, por ayudarme infinitamente a descansar del trabajo. Su único interés en mi bienestar y su inaudita capacidad de discreción respecto a la tesis han constituido mi principal refugio de reposo.

Gracias a todos mis compañeros de despacho y de grupo, que han hecho de mi estancia en la universidad una fantástica experiencia. Gracias en especial a Javi, por ser para mí como un "Michael Slackenerny", un mentor en la vida académica aparte de la investigación; a Jorge, por pelearse duramente con mi software sin que le pagaran por ello, dándome ese punto de vista externo; y a Paco, por esas interesantes e interminables discusiones filosóficas sobre el uso de los términos y la importancia de la reutilización.

Gracias a todos mis amigos, que llevan ya tiempo llamándome "doctor Chema", a pesar de lo mucho que quedaba para que eso fuese una realidad. Gracias a Carlos y a Damiano, compañeros que cualquiera querría tener a su lado en una batalla. Y gracias a Lisa, por cantarme la canción de la felicidad cuando veía tristeza en mi cara.

Gracias a todos de corazón.

Abstract

Every real story is a never ending story.

Mister Koreander to Bastian. The neverending story.

Information in an AmI (Ambient Intelligence) system has diverse natures. Raw data coming from sensors are aggregated and filtered to create more abstract information, which can be processed by application components that observe it to decide what actions should be performed. This process involves several activities: finding the available sources of information and their types, gathering the data from these sources, facilitating the fusion (aggregation and derivation) of the different pieces of data, and updating the representation of this context to be used by applications. The reverse process also appears in AmI systems, when changes in the context representation trigger actions in actuator devices.

In addition, these devices can be added, change or fail, modifying the system topology. Also, other circumstances may change, affecting users' activities, e.g., time, location, or the presence of other users. These changes modify the information the system has available to satisfy users' needs (i.e., the context). AmI systems need to adapt to these evolving conditions in order to be able to provide their services, but being as unobtrusive as possible for their users. There are also performance requirements that a system must fulfill to provide responses to environment stimuli in real time. Opportunistic control mechanisms address these issues by monitoring the context, and suspending or resolving goals when the appropriate conditions are met.

This work presents FAERIE (Framework for AmI: Extensible Resources for Intelligent Environments), a framework that addresses the previous requirements by providing means for the management and fusion of context information at different levels. It is implemented as a distributed blackboard model. Each node of the system has a private blackboard to manage pieces of information that can be accessed by observer components, either locally or remotely (from other blackboards) in a transparent way. It also provides facilities to implement adaptation patterns, such as adaptation to topology changes or to certain activities in a workflow.

This framework is offered with a set of elements to facilitate its use for developing AmI applications. Concretely, a development guide is provided, which explains the process from the extraction of requisites to the implementation and deployment using the architecture. In addition, the framework software comes with a set of tools oriented to make a development with *continuous integration* of the software modules. This concept pursues controlling at every moment the development state and the "health" of the software, among other aspects, via automatic execution and validation of tests, under different deployment and use cases. In order to automatize most of the validation process of an application, an adapter to execute 3D simulations has been developed for the framework. This way, applications can be deployed in a simulated environment as if it was the real one, reducing the costs of error detection in the early stages of testing.

Compared to other architectures, FAERIE offers a more robust and simpler development framework, which gives enough control to handle the complexity of this kind of applications, and reduces the effort of their tracking and validation. Regarding its own architecture, the distributed blackboard model makes context acquisition transparent for its use, maintaining the scalability and avoiding the problem of having a single point of failure. In addition, the context processing model used is less restrictive than in other architectures, allowing each component to access information at any abstraction level.

Resumen

Toda historia es una historia interminable.

El señor Koreander a Bastian. La historia interminable.

La información en un sistema de IAm (Inteligencia Ambiental) es de naturaleza diversa. Los datos sin procesar de los sensores se agregan y filtran para crear información abstracta, que puede ser procesada por componentes que observan sus cambios para decidir qué acciones tomar. Este proceso involucra las siguientes tareas: encontrar las fuentes de información disponibles y sus tipos, reunir los datos proporcionados por estas fuentes, facilitar la fusión (agregación y derivación) de los fragmentos de información, y actualizar la representación de este contexto para que sea usada por aplicaciones. El proceso inverso también aparece en los sistemas IAm, cuando cambios en la representación del contexto disparan acciones en dispositivos actuadores.

Además, los dispositivos pueden ser agregados, cambiados o fallar, modificando la topología del sistema. También, pueden cambiar otras circunstancias afectando a las actividades de los usuarios, p.ej., el tiempo, la localización, o la presencia de otros usuarios. Estos cambios modifican la información que el sistema tiene disponible para satisfacer las necesidades de los usuarios (es decir, el contexto). Los sistemas IAm, deben adaptarse a estas condiciones cambiantes para ser capaz de proporcionar sus servicios, siendo tan poco intrusivos como sea posible. También hay requisitos de rendimiento que el sistema debe cumplir para proporcionar respuestas en tiempo real a los estímulos del entorno. Los mecanismos de control oportunista abordan estos aspectos monitorizando el contexto, y suspendiendo o resolviendo objetivos cuando las condiciones apropiadas se cumplen.

Este trabajo presenta FAERIE (*Framework for AmI: Extensible Resources for Intelligent Environments*), una infraestructura que aborda los requisitos anteriores facilitando la gestión y fusión de información del contexto a distintos niveles. Está implementada siguiendo un modelo de pizarras distribuidas. Cada nodo del sistema tiene una pizarra privada para gestionar fragmentos de información a los que pueden acceder componentes obser-

vadores, ya sea local o remotamente (desde otras pizarras) de una forma transparente. También proporciona facilidades para implementar patrones de adaptación tales como a cambios en la topología o a ciertas actividades en un flujo de trabajo.

La infraestructura se ofrece con un conjunto de elementos para facilitar su uso en el desarrollo de aplicaciones IAm. Concretamente, se proporciona una guía de desarrollo, que explica el proceso desde la extracción de requisitos a la implementación y despliegue siguiendo la arquitectura. Además, la infraestructura software viene con un conjunto de herramientas orientadas a hacer un desarrollo con *integración continua* de los módulos software. Este concepto persigue controlar en todo momento el estadio de desarrollo y la "salud" del software, entre otros aspectos, mediante la ejecución y validación automática de pruebas, bajo distintos despliegues y considerando diferentes casos de uso. Para automatizar en la mayor medida posible todo el proceso de validación de una aplicación, se ha desarrollado un adaptador de la infraestructura para hacer uso de una plataforma de simulación de entornos 3D. De esta manera, las aplicaciones pueden desplegarse en el entorno simulado como si fuese el real, abaratando los costes de detección de errores en las primeras fases de pruebas.

Frente a otras arquitecturas, FAERIE ofrece un marco de desarrollo más robusto y sencillo, que otorga suficiente control para manejar la complejidad de este tipo de aplicaciones, y reduce el esfuerzo del seguimiento y validación de su desarrollo. Con respecto a su propia arquitectura, el modelo de pizarras distribuidas permite que la obtención del contexto sea transparente para su uso, manteniendo la escalabilidad y evitando el problema de tener un único punto de fallo. Aparte de esto, el modelo de procesamiento del contexto utilizado es menos restrictivo que el de otras arquitecturas, permitiendo a cada componente del sistema acceder a información a cualquier nivel de abstracción.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Motivación y objetivos

"Es muy peligroso, Frodo, cruzar la puerta", solía decirme. "Vas hacia el camino y si no cuidas tus pasos no sabes hacia dónde te arrastrarán."

Frodo Bolsón, citando a Bilbo Bolsón.
La Comunidad del Anillo, capítulo 3.

Resumen: En este capítulo se explica en qué consiste el presente trabajo: el contexto y la motivación con la que se realiza, los objetivos que persigue, el marco de investigación en que se encuadra y la metodología utilizada para desarrollarlo. También se describe la estructura del propio documento.

1.1. Introducción

En la última década, especialmente con la popularización de los teléfonos inteligentes o "*smartphones*", se ha asistido a la proliferación e integración en la vida cotidiana de multitud de tecnologías de diversa índole, que proporcionan a muchas personas un acceso casi ubicuo a capacidades avanzadas de computación. Éstas incluyen las comunicaciones inalámbricas (*Bluetooth*, Wi-Fi, 3G, 4G, etc.), la detección de localización (GPS, multilateración GSM, etc.), movimiento y orientación, así como el reconocimiento facial y del habla. Dichas tecnologías se aplican a todo tipo de dispositivos, tales como ordenadores y videoconsolas portátiles, y de forma integrada en distintos entornos, como en vehículos, dentro de locales o en instalaciones (puntos Wi-Fi en autobuses, cafeterías, universidades y quioscos). Además, el avance en los sistemas operativos de estos aparatos ha facilitado el desarrollo de aplicaciones para ellos, dando así acceso en cualquier lugar a servicios

como las redes sociales y la computación en la "nube" (Armbrust et al., 2010). Como resultado de este desarrollo, a día de hoy existe una presencia muy importante de la informática en cualquier ámbito de la vida. En otras palabras, en un espacio cualquiera ocupado por personas, es muy sencillo encontrar al menos un dispositivo electrónico capaz de ejecutar programas bastante complejos.

Lo anterior, unido a la disminución del tamaño y del coste de fabricación del hardware, conforma el escenario perfecto para el desarrollo de la IAm (Inteligencia Ambiental), cuyo objetivo es la integración de dispositivos y programas informáticos de forma transparente como parte del entorno (Weiser, 1993; Remagnino et al., 2005). Aunque no existen definiciones totalmente exactas de este concepto, sus aplicaciones comparten la visión de múltiples elementos sensores, actuadores y computadores, muchos de ellos portátiles y de bajo coste, interconectados y distribuidos en distintos ámbitos para dar asistencia a los usuarios en sus posibles actividades dentro de los mismos. Esto se considera como un avance frente al paradigma tradicional de aplicación "de escritorio", ya que en este caso se considera que la informática es la que se adapta al entorno del usuario, en lugar del usuario al entorno de la informática (York y Pendharkar, 2004).

La asistencia, tal y como la entiende la IAm, implica una mejora de la eficiencia en actividades y flujos de actividades (o flujos de trabajo) en diferentes dominios. En este sentido, abarca desde la ayuda a personas con dificultades en su vida diaria (por ejemplo ancianos, personas con movilidad reducida o enfermos crónicos), hasta la asistencia en circunstancias o contextos muy concretos. Un ejemplo de lo primero sería el uso de sensores biométricos junto con aparatos de uso cotidiano como ordenadores o electrodomésticos (mejorados para este uso), para aprender los hábitos de los usuarios. De este modo se podrían detectar anomalías que puedan significar una emergencia, o coordinar el funcionamiento automático de ciertos dispositivos para facilitar la vida diaria. En el segundo caso, el uso de localización y de reconocimiento y síntesis de voz podría aplicarse en el contexto de una actividad para facilitar información relevante en su desempeño. Por ejemplo, en un congreso científico sería posible guiar al usuario hacia los distintos talleres, y una vez allí ofrecerle información de los ponentes y las personas presentes en la sala. Esto podría hacerse mediante comunicación con los dispositivos móviles habilitados de los usuarios en la sala. Todas estas aplicaciones, llamándose "ambientales", pretenden prestar ayuda al usuario sin interferir excesivamente en el transcurso habitual de las actividades.

Estos sistemas tienen una serie de características que suponen un reto en su desarrollo, y que los hacen interesantes para la investigación en múltiples campos. Una cualidad principal es la de ser capaces de combinar la información de los sensores desplegados en su entorno para inferir las actividades que tienen lugar en el mismo en cada momento, y usar ese conocimiento para

anticipar las necesidades de los usuarios y construir respuestas adecuadas. Aquellos espacios físicos habilitados con estas capacidades son conocidos como *entornos inteligentes*. Estos entornos deben adaptarse al usuario y no al contrario, por tanto, han de ser flexibles y permitir integrar nuevos dispositivos de distintos tipos, así como tolerar fallos sin un esfuerzo notorio de configuración por parte de los usuarios. Existen de este modo tres requisitos fundamentales: la capacidad de conocer y abstraer información sobre los usuarios en tiempo real a partir de sensores; el uso de esta información de forma adaptativa para conseguir ciertos objetivos; y la tolerancia a cambios en la topología de los entornos en tiempo de ejecución. Estos tres requisitos suelen resumirse como "ser sensible al *contexto*" ("*context-awareness*"), tanto del usuario (actividades y situación actual), como del sistema (recursos disponibles y topología). El concepto de *contexto* más comúnmente utilizado en la literatura se refiere a cualquier información relacionada con personas, lugares u objetos que es relevante para la función de las aplicaciones de IAm (Abowd et al., 1999). Esta definición se explicará en mayor detalle en el Capítulo 2.

A la hora de implementar los requisitos mencionados, surgen múltiples retos: modelar los distintos elementos de una aplicación y su "contexto"; definir cómo se gestiona la información contextual, es decir, su obtención y tratamiento; y finalmente, presentar una capa de uso para las aplicaciones que permita la adaptación automática a este contexto. Además, de forma paralela, han de generarse una serie de mecanismos que faciliten el desarrollo de las aplicaciones teniendo en cuenta varios aspectos: la importancia de la modularización y el comportamiento dinámico de los componentes, el enorme coste de validación de las aplicaciones en este tipo de escenarios, y la multitud de tecnologías subyacentes que pueden estar involucradas.

Ante los retos planteados, se requiere una arquitectura que los aborde e integre las potenciales soluciones a ellos. El resultado ha de ofrecer una terminología y un marco conceptual para poder modelar las aplicaciones; una infraestructura distribuida que facilite la adquisición de la información y la agregación e interpretación dinámica de la misma para obtener conocimiento; y una serie de facilidades para la implementación de mecanismos de adaptación a los cambios en la información. Paralelamente, deben ofrecerse unas herramientas y guías de uso que cumplan con los requisitos relativos al desarrollo que se han mencionado.

En este trabajo se ha creado la arquitectura FAERIE (*Framework for Ambient intelligence: Extensible Resources for Intelligent Environments*) que aborda estos requisitos, con el propósito de facilitar el desarrollo de aplicaciones de IAm. Además, esta arquitectura en sí misma supone una fuente de investigación posterior, ya que el campo de la IAm integra tecnologías de muy diversa índole, que suelen ir asociadas a problemas que requieren el uso de distintas disciplinas. Por este motivo, el trabajo se convierte en un "banco

de pruebas" con distintos puntos en los que situar el foco de la investigación. En la Sección 1.2 se explica con detalle cuáles son los objetivos concretos que se han planteado para este trabajo. Posteriormente, en la Sección 1.3 se describe qué metodología se ha seguido para llevarlo a cabo.

1.2. Objetivos

A continuación se detallan los objetivos del presente trabajo. En primer lugar se observan desde un punto de vista tecnológico y de investigación, es decir, cuestiones que ayuden a afrontar los retos mencionados del campo de estudio. En segundo lugar se consideran los objetivos desde el punto de vista del marco de desarrollo. Se trata de aquellas cuestiones que están orientadas a crear un proceso de generación de aplicaciones de este tipo.

1.2.1. Desde un punto de vista tecnológico

La investigación se dirige a las arquitecturas de sistemas IAm y a infraestructuras para su desarrollo. El objetivo es desarrollar la arquitectura FAERIE, como marco de trabajo para facilitar la creación de este tipo de aplicaciones. Los objetivos concretos son los siguientes:

- Crear un marco conceptual para sistemas de IAm. Esto es, crear un modelo de los elementos identificables y relevantes en un sistema y cómo se van a comportar y relacionar entre sí.
- Establecer una infraestructura de desarrollo bajo el paradigma de "sensibilidad al contexto". Se trata de implementar los mecanismos fundamentales comunes para así reducir el tiempo de desarrollo de estas aplicaciones y conseguir enfocarse en cuestiones de lógica, apartándose de los detalles de bajo nivel. Se han de considerar despliegues distribuidos y dinámicos en la gestión del contexto. Eso quiere decir que la arquitectura y la infraestructura deben ser capaces de mantener una consistencia en la gestión del contexto bajo cambios en su topología en tiempo de ejecución, y además deben ser capaces de trabajar de forma distribuida.
- Proporcionar patrones y mecanismos para implementar funcionalidad de adaptación automática al contexto. Para aquella que no se pueda ofrecer de forma totalmente automática, dar patrones claros para implementarla basándose en la infraestructura.

1.2.2. Desde un punto de vista del marco de desarrollo

La infraestructura resultante debe ofrecerse de tal manera que facilite su desarrollo y validación. Para ello se han considerado los objetivos que se

enuncian a continuación:

- Hacer uso del desarrollo con *Integración Continua*. Este paradigma persigue mejorar el ciclo de desarrollo mediante herramientas que faciliten realizar cambios de forma incremental, pudiendo evaluar la calidad de cada una de las partes del software. Tecnologías que se utilizan en este paradigma son, por ejemplo, las pruebas unitarias, software de gestión de módulos, y servidores de integración y de control de versiones. Estas herramientas permiten también crear de forma automática elementos desplegables para distintas configuraciones, lo cual resulta muy útil en un contexto en el que pueden existir múltiples dispositivos distribuidos.

- Dar soporte a la validación supervisada de los requisitos mediante simulación. Ofrecer, dentro del ciclo de desarrollo, la posibilidad de probar las aplicaciones en un entorno 3D simulado. Este objetivo es especialmente interesante teniendo en cuenta el enorme coste de implementar despliegues reales. Esta utilidad sirve tanto para encontrar problemas en el despliegue planificado antes de realizar el despliegue real, como de herramienta para que el cliente pueda validar la solución.

- Establecer guías de uso de la arquitectura. Esto es, indicar claramente cómo hacer un uso correcto de los conceptos y mecanismos de la infraestructura para obtener los resultados deseados. Estas guías se pueden encuadrar en diferentes metodologías, ya que más que un procedimiento de desarrollo, han de ser una lista de cuestiones y de tareas necesarias para el correcto uso de las herramientas. El resultado será un conjunto de tutoriales (Fernández-de Alba, 2013) y manuales de uso de la infraestructura (ver Capítulo 9).

1.3. Marco de la investigación

El presente trabajo se presenta como proyecto para aspirar al título de Doctor en Informática, dirigido por los doctores Juan Pavón Mestras y Rubén Fuentes Fernández, dentro de las líneas de trabajo del grupo de investigación GRASIA (GRupo de investigación en Agentes Software: Ingeniería y Aplicaciones), de la Universidad Complutense de Madrid. Estas líneas abarcan la Ingeniería del Software Orientada a Agentes, la Inteligencia Ambiental, la Ingeniería Dirigida por Modelos y la Simulación Social, entre otras.

El proyecto se encuadra principalmente dentro del campo de la Inteligencia Ambiental. De esta forma, las publicaciones relacionadas se aportan como resultados del proyecto nacional de investigación SociAAL (*Social Ambient-Assisted Living*), financiado por el Ministerio de Economía y Competitividad (con identificador TIN2011-28335-C02-01). El proyecto ha sido concedido al grupo GRASIA en conjunción con el grupo ANTS (*A Non sTop workerS*)

de la Universidad de Murcia. Este proyecto tiene como objetivo desarrollar un lenguaje de modelado de sistemas de Inteligencia Ambiental, que tenga en cuenta las necesidades socioculturales de los beneficiarios de los sistemas, además de infraestructura para utilizarlo en desarrollos. En concreto, se plantean casos de estudio orientados a ayudar a enfermos de Parkinson en sus actividades cotidianas.

Como muestra de las sinergias entre los grupos participantes en este proyecto, hay que señalar que algunos trabajos del grupo ANTS relacionados con la IAm han servido de referencia y otros han sido utilizados directamente en este trabajo. Este es el caso de la herramienta UbikSim, que sirve para modelar y simular entornos inteligentes en 3D (Campuzano et al., 2011).

Finalmente, el autor de este trabajo ha disfrutado durante su desarrollo de financiación por parte de una beca FPU (Formación de Profesorado Universitario), del Ministerio de Educación, Cultura y Deporte, con el fin de conseguir el título de Doctor en Informática.

1.4. Metodología

El proceso de investigación ha constado de una primera fase intensiva de estudio de la literatura, donde se han analizado los trabajos existentes en este campo (que serán explicados en detalle en el próximo capítulo). De este estudio se han extraído las características fundamentales tanto de las casos de estudio utilizados, como de las arquitecturas de los sistemas utilizados para resolverlos. Se han identificado también los puntos de diferenciación existentes entre las arquitecturas, así como sus ventajas y desventajas.

La hipótesis asumida en este trabajo consiste en que los problemas a resolver por la arquitectura objetivo pueden irse identificando de forma incremental definiendo casos de estudio de complejidad creciente a base de ir agregando nuevas características a casos de estudio más simples. Estos casos servirían también a su vez para validar las soluciones propuestas. Así, la segunda fase ha consistido en el desarrollo de la arquitectura siguiendo este método, haciendo uso de las distintas alternativas consideradas deseables de las arquitectura estudiadas. Para esto se ha seguido el siguiente proceso iterativo e incremental:

1. Definir un caso de estudio acorde a las características típicas de aquellos planteados en la literatura, con una cantidad mínima de características.

2. Ampliar/modificar el marco conceptual de la arquitectura para modelar el contexto y los elementos del caso de estudio.

3. Agregar/modificar las capacidades necesarias en la infraestructura para automatizar la gestión de los comportamientos presentes en el caso de

estudio. Considerar para esto alternativas existentes en el estado del arte que supongan ventajas deseables.

4. Reimplementar la aplicación haciendo uso de estas nuevas capacidades.
5. Extraer posibles patrones repetibles de la aplicación y abstraerlos como nuevos mecanismos.
6. Probar la aplicación de nuevo y validar.
7. Si el resultado es satisfactorio, agregar nuevas características al caso de estudio.
8. Volver al punto 2.

1.5. Estructura del documento

En los dos primeros capítulos se presenta la introducción y el origen del trabajo. En el presente Capítulo 1 se explican la motivación y los objetivos del trabajo, así como el marco en el que surge y su metodología de desarrollo. En el Capítulo 2 se describe el contexto en el que se encuadra dentro de la literatura de los campos que abarca.

En los siguientes capítulos se describe el contenido concreto del trabajo. Comenzando con el Capítulo 3, se ofrece una visión de las decisiones de partida con respecto a un conjunto de requisitos identificados, seguida del Capítulo 4, donde se muestra una visión global de la arquitectura y de cómo se interrelacionan sus partes. A partir de ahí, en el Capítulo 5 se introduce la capa básica de la infraestructura, esto es, aquella que define los componentes fundamentales, los de menor nivel de abstracción, y cómo se interconectan unos con otros. Después, en el Capítulo 6 se describen los mecanismos que se utilizan para gestionar la información del contexto usando los componentes de la capa básica. En el Capítulo 7 se muestran distintos patrones de control para permitir la adaptación automática al contexto por parte de las aplicaciones. Finalmente, en el Capítulo 8 se estudia el uso e integración en la infraestructura de agentes inteligentes, así como sus posibilidades y ventajas.

En los capítulos siguientes se cuenta el modo de utilización y las experiencias de uso de la arquitectura. En el Capítulo 9, se describe el proceso de desarrollo de un SSC (Sistema Sensible al Contexto) haciendo uso de FAERIE, y las herramientas que se usan como asistencia. Después, en el Capítulo 10 se explica de qué modo se integra el desarrollo con el uso de entornos virtuales 3D. Por último, en el Capítulo 11 se definen dos casos de estudio como ejemplo de uso concreto de la arquitectura.

Finalmente, en el Capítulo 12 se resumen una serie de conclusiones extraídas de todo el trabajo y se plantean líneas de investigación derivadas de esta tesis.

Capítulo 2

Estado del arte

—¿Has averiguado lo del libro?
—No te lo vas a creer, tenemos que volver a 1955.
—¡No me lo puedo creer!

Doctor Emmett Brown y Marty McFly.
Regreso al futuro II.

Resumen: En este capítulo se estudian algunos de los trabajos más importantes en este área. Finalmente, se extraen las características de estos trabajos y se analizan de forma cualitativa.

2.1. Introducción

Los sistemas de IAm pueden abarcar numerosos ámbitos de aplicación, como se ha señalado en la introducción de este trabajo. En cada uno de estos ámbitos puede ser conveniente la utilización de unas u otras de las tecnologías usadas en el área, las cuales traen consigo sus propios problemas y campos de investigación independientes. Además, existe un rango muy amplio de niveles de abstracción, desde el control directo de dispositivos sensores y actuadores, hasta la inferencia del contexto, actividades, relaciones entre las entidades, etc.

La variedad anterior da lugar a dos dimensiones distintas en las que considerar los campos de estudio: tecnologías y niveles de abstracción de las mismas. En la primera dimensión se encuentra el estudio de las tecnologías usadas (p.ej., para el reconocimiento del habla, localización, identificación visual, o comunicación de dispositivos). En la segunda dimensión las soluciones se clasifican según los niveles de abstracción donde ofrecen el grueso de sus servicios. Aquí se encuentran desde los estudios situados en las capas

más bajas del desarrollo (tratando directamente con dispositivos y sus controladores, protocolos de conexión y comunicación, etc.), hasta las capas más altas, donde aparecen aspectos de inferencia de información abstracta (fusión de información, aprendizaje, gestión de flujos de trabajo, etc.), pasando por el estudio de las arquitecturas que permiten su unión. Este trabajo va a enfocarse en la segunda dimensión.

Debido a la diversidad de aspectos que pueden abarcar, comparar directamente dos trabajos puede resultar muy complejo. Para facilitar este estudio, a continuación se van a analizar concretamente los elementos arquitectónicos que permiten realizar la gestión de información de contexto a partir de información de bajo nivel. Esto nos sitúa en el campo de las aplicaciones "sensibles al contexto" o "*context-aware*", como ya se mencionó en el capítulo anterior (Baldauf et al., 2007). A continuación se presentarán algunos de los proyectos más relevantes en este campo, dado su impacto. Estos trabajos clave identifican las características esenciales de los sistemas sensibles al contexto. Después de esto, se hará un análisis de cada una de estas características de forma independiente, para localizar cuáles son las distintas alternativas para las mismas.

2.2. Estudio de trabajos del área

Esta sección describe los trabajos más relevantes en el ámbito de los SSC, identificando los aspectos fundamentales de las distintas arquitecturas propuestas. Posteriormente se analizan comparativamente sus características en la sección siguiente.

2.2.1. Ranganathan y Campbell (2003)

Este trabajo propone una infraestructura que facilita el desarrollo de *agentes sensibles al contexto*: permite a los agentes **adquirir y modelar información contextual**, **procesarla y razonar** sobre ella usando diferentes lógicas (como lógica de primer orden y lógica temporal), y **adaptarse** al contexto cambiante. Pertenece a una infraestructura mayor llamada *Gaia*, que sirve para habilitar espacios inteligentes. Proporciona servicios tales como difusión de eventos y descubrimiento e identificación de recursos o entidades del entorno. Estos servicios son llevados a cabo también por agentes.

Para **adquirir el contexto**, los agentes se comunican con los recursos disponibles del entorno. Después, **modelan** este contexto utilizando ontologías escritas en DAML+OIL (Lenguaje de Marcado de Agentes de DARPA + Lenguaje de Intercambio de Ontologías, *DARPA Agent Markup Language + Ontology Interchange Language*) (van Harmelen et al., 2001; Ranganathan y Campbell, 2003), permitiendo así la posible interoperabilidad con agentes externos. De este modo la información del contexto se modela mediante pre-

dicados lógicos, siguiendo el paradigma de la lógica descriptiva. Esto significa que cada propiedad del contexto es un predicado distinto, al que se le aplican distintos argumentos. Un predicado enuncia una determinada propiedad contextual que es cierta. La consistencia de una representación puede validarse haciendo uso de razonadores sobre la ontología. Esta infraestructura usa CORBA (*Common Object Request Broker Architecture*) (Pope, 1998) para habilitar la comunicación entre los agentes distribuidos.

Los agentes pueden **razonar** sobre el contexto y **aprender** distintos comportamientos según la situación, usando por ejemplo aprendizaje Bayesiano y aprendizaje por refuerzo. También pueden especificarse estos comportamientos de forma estática mediante reglas. Los tipos de agentes que forman la **arquitectura** en un sistema de este tipo son los siguientes:

Context Providers Son sensores o fuentes de datos contextuales. Pueden ser consultados o dejar eventos en un canal específico.

Context Synthesizers Abstraen información contextual para obtener nueva información.

Context Consumers Obtienen información contextual para llevar a cabo distintos comportamientos.

Context Provider Lookup Service Es el servicio donde los *Context Providers* publican el contexto que proporcionan para que el resto de agentes los encuentren.

Context History Service Mantiene el registro de cambios en el contexto.

Ontology Server Mantiene las ontologías que se utilizan en el sistema.

2.2.2. Gu et al. (2005)

Propone una infraestructura basada en servicios para la construcción y prototipado de servicios sensibles al contexto, que proporciona soporte para **adquirir**, **modelar** e **interpretar** varios contextos. Se integra dentro de una arquitectura distribuida llamada SOCAM (*Service-Oriented Context-Aware Middleware*), basada en su propio modelo del contexto. Este modelo convierte varios espacios físicos, desde los que se adquieren los contextos, en un único espacio semántico, donde los contextos son compartidos por los servicios sensibles al contexto. La comunicación entre los componentes se realiza mediante Java RMI (*Remote Method Invocation*) (Plasil y Stal, 1998).

El **modelo** de contexto anterior se basa en un modelo formal del contexto basado en ontologías usando OWL (*Web Ontology Language*) (Hawkinson, 1975) para abordar el razonamiento, la clasificación del contexto y sus dependencias. La información del contexto se representa como predicados de

primer orden de la forma: $Predicado(Sujeto, Valor)$. El paradigma seguido es la lógica descriptiva, al igual que con DAML+OIL.

Este trabajo adopta un enfoque de dos capas para diseñar las ontologías del contexto: una ontología general para todos los dominios de sistemas ubicuos, y una específica de dominio con conceptos más concretos. De esta manera, el desarrollo y procesamiento de cada contexto específico es independiente. En el momento de su uso, las ontologías de bajo nivel se "autoenlazan" con la de nivel superior, pudiendo así aprovechar el conocimiento más abstracto que les sea aplicable. Las dependencias entre los distintos tipos de información contextual están sujetas a aprendizaje, de tal modo que puede establecerse la probabilidad de que cierta dependencia exista en un determinado momento. Esto permite tratar más fácilmente con la posible falta de información.

La **arquitectura** consta de los siguientes componentes:

Context Providers Abstraen contextos útiles, internos o externos, y los convierten en representaciones OWL.

Context Interpreter Proporciona servicios de razonamiento para procesar la información contextual. Tiene una base de conocimiento que contiene un conjunto de reglas e información para facilitar esta tarea.

Context Database Almacena ontologías y contextos para un subdominio. Hay una por cada dominio.

Context-Aware Services Hacen uso de diferentes niveles de contexto y adaptan la manera de comportarse de acuerdo al mismo. Se adaptan mediante reglas.

Service Locating Service Proporciona un mecanismo donde los proveedores de contexto y el intérprete pueden anunciar su presencia.

2.2.3. Bardram (2005)

Describe la JCAF (*Java Context-Aware Framework*), que es una infraestructura sensible al contexto basada en Java para crear aplicaciones de este tipo. Su propósito es ser simple y robusta, para que los programadores puedan extenderla. Proporciona las interfaces mínimas para ser lo más generalista posible.

La infraestructura es un sistema distribuido basado en la idea de dividir la **adquisición del contexto**, su **modelado** y **gestión**, y su **distribución** en una red de servicios de contexto. Estos servicios cooperan de forma P2P (*Peer to Peer*) mediante detección de eventos, de tal manera que cada uno se ocupa independientemente de su entorno cercano. Además, la inclusión de nuevos servicios se hace de manera dinámica, mediante mecanismos basados en el uso de Java RMI (Plasil y Stal, 1998).

El modelo de programación de JCAF promueve el uso de medidas de calidad para la información contextual (por ejemplo la incertidumbre de la información). Este modelo especifica cómo usar la API (*Application Programming Interface*) de los servicios de contexto, cómo modelar las entidades, y cómo usar la infraestructura basada en eventos. Proporciona un diagrama de clases y las operaciones más típicas.

La **arquitectura** se divide en tres capas de abstracción:

Capa de Clientes del Contexto Contiene las aplicaciones sensibles al contexto. Éstas pueden agregar, consultar o utilizar los transformadores del contexto, que son aquellos servicios que abstraen la información de bajo nivel proveniente de los sensores y "detallan" la información que va a los actuadores.

Los clientes pueden obtener la información haciendo peticiones ("request") o suscribiéndose a los cambios de las entidades.

Capa de Servicios del Contexto Cada servicio (análogo a un Web Service) tiene un contenedor de entidades y un repositorio de transformadores del contexto.

Una entidad es un objeto Java que se ejecuta dentro del contenedor del contexto en el que se ha añadido, el cual controla su ciclo de vida. Representa a algún objeto físico o virtual y responde a sus cambios, manteniendo un estado. Además, maneja a sus suscriptores, notificándoles los cambios en su estado.

Estas entidades tienen acceso al "*Entity Environment*", que les proporciona a su vez acceso a diversas utilidades, objetos compartidos, API, y a los transformadores de contexto. Un transformador del contexto es capaz de calcular nueva información a partir de aquella existente en el contexto, mediante inferencia o aprendizaje.

También contiene un control de acceso que usa "*Java Authentication and Authorization Service*", para asegurar autentificación a los clientes del servicio.

Capa de Monitores y Actuadores del Contexto Los monitores y actuadores se consideran clientes del contexto. Los monitores colaboran para recoger información de un sensor y asociarla a una entidad. Los actuadores colaboran con los dispositivos correspondientes para "cambiar" el contexto.

El framework puede manejar la adquisición de información, asíncronamente (mediante notificación), o síncronamente (bajo petición).

2.2.4. Henricksen et al. (2004)

Presenta un marco conceptual y una infraestructura que trata de simplificar las tareas de diseño e implementación asociadas con el software sensible al contexto y de facilitar el rápido prototipado y experimentación del mismo.

Uno de sus objetivos es proporcionar abstracciones del contexto bien definidas, aparte del soporte de la infraestructura, por lo que centra su desarrollo en el **modelado del contexto**. Para ello presenta tres enfoques de modelado para soportar: 1) la exploración y especificación de los requisitos de contexto de una aplicación, 2) la **gestión** de la información del contexto almacenada en un repositorio, y 3) la especificación de clases abstractas que estén cerca de la forma de pensar del programador. Estos tres enfoques son:

Representación Relacional Aprovecha el mapeo ORM (*Object-Relational Model*) para conseguir una representación relacional del contexto. Ésta se emplea para capturar las restricciones de su lenguaje, guardar información y consultarla. Es un modelo de predicados muy similar a los utilizados en otros trabajos.

Modelado Gráfico propone un lenguaje de modelado del contexto para asistir en la tarea de explorar los requisitos. Se define como una extensión de ORM. Clasifica los tipos de contexto, la información sobre calidad, y las dependencias. También permite anotar hechos para indicar si se permite información ambigua, y soporta ciertas restricciones.

Situación Abstracta Este enfoque es diferente a los dos anteriores, los cuales definen el contexto con muy baja granularidad. Para evitar estos inconvenientes, se definen las *situaciones abstractas*, que son fórmulas lógicas más elaboradas para describir hechos del contexto.

Las tres aproximaciones de modelado del contexto tienen en cuenta la incertidumbre de la información y las posibles inconsistencias entre distintas fuentes del mismo tipo de contexto.

Aparte, el trabajo presenta dos enfoques para el **procesamiento del contexto**, que son:

Modelo de Saltos las condiciones del contexto modifican la puntuación de distintos cursos de acción. Al final de la evaluación, se llevan a cabo las acciones mejor puntuadas.

Detonación las acciones se eligen entre un conjunto de alternativas según las condiciones del contexto en el momento en el que este cambia.

A partir de los elementos anteriores, propone una infraestructura software por capas:

Capa de recogida obtiene la información del contexto y lleva a cabo tareas de agregación e interpretación. El acoplamiento con los componentes a los que notifican estas tareas es bajo gracias al modelo de eventos.

Capa de recepción mapea las entradas proporcionadas como eventos por la capa anterior a uno de los modelos descritos anteriormente.

Capa de gestión es responsable de mantener un conjunto de modelos del contexto y sus instanciaciones. La intención es que esta capa sea distribuida, pero no lo tienen implementado.

Capa de consultas permite hacer consultas sobre la información contextual y proporciona notificaciones de cambios.

Capa de adaptación contiene repositorios comunes de preferencias, situaciones y detonadores.

Capa de aplicación proporciona soporte para los dos modelos de procesamiento mencionados anteriormente.

2.2.5. Chen et al. (2003)

Describe CoBrA-ONT (*CoBrA Ontology*), un conjunto de ontologías para **modelar** el contexto expresadas en OWL (Hawkinson, 1975), y un motor de inferencia para **razonar** con la ontología dada. Se desarrolla como parte de una **arquitectura** "centrada en bróker" llamada CoBrA (*Context Broker Architecture*), que proporciona compartición de conocimiento, razonamiento sobre el contexto y protección de privacidad.

Un elemento central de esta arquitectura es la presencia de un agente inteligente llamado "bróker del contexto". Es un servidor de información contextual especializado que corre en una máquina estacionaria del entorno. Su rol es mantener un modelo compartido del contexto en nombre de una comunidad de agentes y dispositivos en el espacio, y proteger la privacidad de los usuarios asegurando el cumplimiento de las políticas definidas cuando comparte información. Todas las entidades computacionales tienen conocimiento del bróker del contexto y se comunican con él utilizando el lenguaje FIPA-ACL (*Foundation for Intelligent Physical Agents - Agent Communication Language*) (Steiner, 1998).

El agente bróker contiene los siguientes componentes:

Base de Conocimiento almacenamiento persistente del contexto.

Motor de Razonamiento un motor de inferencia reactivo.

Módulo de Adquisición una librería de procedimientos que forma una abstracción middleware para **adquirir** conocimiento.

Módulo de Gestión de Directivas un conjunto de reglas de inferencia que deduce instrucciones para decidir los permisos correctos para compartir información contextual y para seleccionar los receptores de notificaciones.

La carga del procesamiento se realiza en el bróker con el fin de permitir la existencia de componentes del sistema en dispositivos con recursos muy limitados. Además, esto permite una mayor seguridad. Sin embargo, se declara que este enfoque puede producir un cuello de botella. Para evitarlo se propone crear una "federación de brókers", que se organice para sincronizar la información contextual.

2.3. Análisis de características

A la vista de los distintos trabajos examinados, que constituyen los ejemplos más representativos del campo, se aprecia que todos ofrecen una manera de modelar el contexto que manejan, y una infraestructura que permite gestionarlo. Esta gestión consiste principalmente en la obtención de la información del entorno, su procesado mediante razonamiento, y su distribución a los distintos componentes. Además, muchos trabajos integran esta infraestructura en una arquitectura mayor que permite el uso del contexto para llevar a cabo comportamientos adaptativos y tareas más complejas.

Atendiendo a los aspectos previos, esta sección va a estructurarse como sigue. En la Sección 2.3.1, se van a tratar las características de la gestión del contexto mencionadas. Después, en la Sección 2.3.2 se van a abordar los mecanismos de adaptación que éstas habilitan. Este análisis va a enfocarse a detectar cuáles son las distintas posibilidades en cada aspecto, para identificar las ventajas y desventajas de cada elección.

Finalmente, se estudia la integración de agentes inteligentes como usuarios de la arquitectura.

2.3.1. Gestión del contexto

Existen múltiples y heterogéneos enfoques para desarrollar SSC. Esto se debe en parte a la generalidad de las definiciones de *contexto* y *sensibilidad al contexto*, que hace que aplicaciones muy distintas puedan alegar ser sensibles al contexto, y al amplio rango de mecanismos, tecnologías y requisitos que aparecen en tales sistemas. Esta sección trata de cubrir este amplio panorama a través de las siguientes subsecciones. Cada una discute algunas de las principales características usadas para evaluar SSC. Tal y como han ido surgiendo en los trabajos estudiados, los cuales corresponden a los más citados del área de estudio, estas características son: **obtención del contexto** (Sección 2.3.1.1), **modelado del contexto** (Sección 2.3.1.2), **procesamiento**

Sistemas / Infraestructuras	**Obtención del Contexto**	**Modelado del Contexto**	**Procesamiento del Contexto**	**Distribución / Organización**
iRoom (Winograd, 2001)	pizarra	tuplas clave-valor	centrado en datos	servidor central / dinámica
CoBrA (Chen et al., 2003)	servidor de contexto	basado en ontologías	centrado en datos	p2p / dinámica
Context Toolkit (Dey et al., 2001)	widgets	tuplas clave-valor	centrado en procesos	p2p / estática
Context Management Framework (Korpipaa et al., 2003)	pizarra	basado en ontologías	centrado en datos	servidor central / dinámica
Hydrogen (Hofer et al., 2003)	servidor de contexto / punto-a-punto	orientado a objetos	centrado en procesos	servidor central / estática
Haya et al. (2006)	pizarra / grafo del contexto	tuplas clave-valor	centrado en datos	servidor central / dinámica
OCP (Nieto et al., 2006)	pizarra virtual	basado en ontologías	basado en reglas	p2p / estática
Henricksen et al. (2006)	servidor de contexto	basado en ontologías	centrado en procesos	p2p / estática
Ejigu et al. (2008)	servidor de contexto	basado en ontologías	centrado en procesos	p2p / estática
Hermes (Buthpitiya et al., 2012)	servidor de contexto	orientado a objetos	centrado en procesos	p2p / dinámica
Venturini et al. (2012)	broker del contexto	basado en ontologías	centrado en procesos	p2p / dinámica
FAERIE	pizarra	orientado a objetos	centrado en datos	p2p / dinámica

Tabla 2.1: Tabla de comparación de características arquitectónicas entre propuestas AmI

del contexto (Sección 2.3.1.3), y **distribución de la arquitectura** (Sección 2.3.1.4). La tabla 2.1 resume los resultados. En esta se incluye FAERIE con carácter ilustrativo, pero sus características específicas se describirán en los capítulos sucesivos.

2.3.1.1. Obtención del contexto

Los SSC están empotrados en los entornos que necesitan conocer, los cuales perciben mediante sensores. Sin embargo, la mayoría de los componentes de estos sistemas no acceden directamente a los sensores, sino que delegan esta tarea en algún mediador. Este mediador organiza el acceso a la información para garantizar, por ejemplo, igualdad y seguridad a las peticiones. El tipo de mecanismo utilizado para esta *obtención del contexto* constituye la primera dimensión de análisis. Hay dos tipos principales de aproximaciones a la obtención del contexto: orientada a *interfaz* y orientada a *datos*.

En la obtención de contexto orientada a *interfaz*, el componente cliente obtiene una interfaz a algún tipo de componente servidor y la utiliza para pedir y obtener información. Esto puede hacerse usando *widgets* o servidores de contexto.

Un *widget* es un componente software que proporciona una interfaz pública local a un sensor físico. Este es el enfoque por ejemplo del *Context Toolkit* (Dey et al., 2001). El *widget* no implementa funcionalidad para acceso remoto, por lo que necesita componentes adicionales para usarse en entornos distribuidos. Como ventaja, proporciona un uso eficiente (i.e, con una carga de gestión baja) siempre que no haga falta acceso concurrente.

Un *servidor de contexto* agrega los servicios de múltiples proveedores de contexto (Ejigu et al., 2008; Buthpitiya et al., 2012; Hong y Landay, 2001). Los consumidores de contexto solicitan a tal servidor de contexto un servicio capaz de proporcionar la información contextual que necesitan. Cuando obtienen el servicio, interaccionan con éste para obtener la información. Una variante de este mecanismo es el *bróker de contexto*, que juega el mismo rol, pero da acceso a agentes proveedores en lugar de a servicios (Venturini, Carbó y Molina, 2012; Chen, Finin y Joshi, 2003). Este enfoque con servidores de contexto puede ocultar las diferencias entre información remota y local, al contrario de lo que ocurría con los *widgets*. No obstante, su uso requiere procesamiento adicional, ya que los servidores de contexto han de manipular el servicio para proporcionar la información solicitada por los clientes.

El otro enfoque principal para obtención de contexto son los mecanismos *orientados a datos*. Estos son soportados por un repositorio central de información a donde acceden los componentes y solicitan datos específicos. Los *modelos de pizarra* pertenecen a esta categoría.

Una pizarra soporta una obtención del contexto indirecta: un componente hace una petición a la pizarra, generando una coordinación de componentes de forma oportunista para proporcionar esta información, que se hace accesible para el cliente en la pizarra cuando está disponible. El uso de este modelo desacopla a los proveedores de contexto de los consumidores, facilitando la reconfiguración del sistema. Como consecuencia de ello, el sistema es más robusto, siempre que el servidor de contexto esté activo. *iRoom* (Winograd, 2001) es uno de los primeros intentos de aplicar el modelo de pizarra a este campo. Haya et al. (2006) presentan un trabajo similar, mejorado con la posibilidad de acceder al contexto navegando a través de las relaciones en el grafo del modelo de contexto. La principal desventaja de este enfoque es la eficiencia, ya que cada comunicación en el sistema ha de pasar a través del servidor de contexto. La OCP (*Open Context Platform*) (Nieto et al., 2006) tiene en cuenta este aspecto. Puede trabajar sin la existencia de un servidor de contexto central, haciendo uso de un espacio de tuplas que conforma una *pizarra virtual*. El espacio de tuplas asigna un identificador único a cada elemento del contexto. Este identificador también se usa como dirección

para encontrar el manejador que gestiona esta información. De esta manera, cuando un componente solicita un determinado elemento de contexto, la petición es redirigida al componente correspondiente, que proporciona el valor directamente. Esta plataforma puede trabajar también usando un servidor central. En cualquier caso, si por algún motivo el componente manejador no se encuentra, la petición es redirigida al servidor existente para asignar un nuevo manejador.

Los enfoques orientados a interfaz y a datos, también se diferencian en la localización del mecanismo para mantener la consistencia. En el primer caso, el cliente elige entre diferentes interfaces, de modo que tiene la responsabilidad de tratar con redundancia e inconsistencia. Para esa tarea, puede hacer uso de heurísticas especificas del dominio. En el segundo caso, dado que el cliente no elige el proveedor del contexto, la responsabilidad de tratar con la redundancia e inconsistencia depende de la infraestructura. Esto libera al cliente de esta gestión, pero en cambio, hace esta gestión menos eficiente, ya que las soluciones tienden a ser más generales.

2.3.1.2. Modelado del contexto

Una vez que se ha obtenido el contexto, el sistema necesita representarlo de una forma adecuada a su funcionalidad. La manera en que lo hace constituye la segunda característica, el *modelado del contexto*. Hay diferentes maneras de implementar el modelo del contexto (Strang y Linnhoff-Popien, 2004): basado en ontología, orientado a objetos y con tuplas clave-valor. Estos enfoques son en ocasiones mejorados con esquemas de marcado o proposiciones lógicas para proporcionar semántica adicional al modelo. Algunas infraestructuras también proporcionan formas gráficas de especificar los tipos de contexto. Henricksen et al. (2006) presentan una como extensión del mapeo objeto-relacional, y Venturini, Carbó y Molina (2012) integran el modelo de contexto con el modelo BDI (*Belief-Desire-Intention*) de la teoría de agentes software (Bratman, 1999).

La representación más simple del contexto es posiblemente como tuplas clave-valor. Un único elemento de contexto se representa como una lista de valores numéricos y de cadena de caracteres, cada uno etiquetado con una clave para indicar el tipo de información que representa. El *Context Toolkit* (Dey et al., 2001) es un ejemplo de este enfoque. La principal ventaja de esta representación es su simplicidad, tanto de uso como de extensión. Sin embargo, esta simplicidad implica capacidades limitadas para trabajar con la información del contexto (por ejemplo, no se pueden definir tipos estructurados o herencia).

Los enfoques basados en ontologías son los más populares por su expresividad y extensibilidad. CoBrA (Chen et al., 2003), la OCP (Nieto et al., 2006), y los trabajos de Gu et al. (2005), Ejigu et al. (2008) y Venturini, Carbó y Molina (2012) proponen modelado basado en ontologías para la

información del contexto. Estos trabajos requieren integrar un parser y un motor para usar ontologías, con lo que demandan más recursos que otros enfoques. Además, las potentes capacidades de razonamiento de las ontologías no se requieren en muchos sistemas, aunque puedan ser deseables en determinados casos. Por esta razón, numerosos trabajos adoptan alguna de las otras alternativas más simples.

El enfoque orientado a objetos para modelar el contexto es también ampliamente usado, por ejemplo en la JCAF (Bardram, 2005) y el proyecto *Hermes* (Buthpitiya et al., 2012). Este enfoque permite aprovechar tecnologías existentes para lenguajes orientados a objetos, tales como serialización de objetos o mapeo a bases de datos relacionales para almacenamiento. De este modo, trae un ahorro importante en formación para equipos de desarrollo y en obtención de herramientas, y reduce el coste relacionado con la heterogeneidad de tecnologías en proyectos.

2.3.1.3. Procesamiento del contexto

La información en crudo, tal y como se obtiene directamente de los sensores, raramente es utilizada por los componentes sensibles al contexto. Esta información normalmente pasa a través de procesos de fusión (agregación o interpretación) para generar una representación en términos del dominio de la aplicación. Esto se hace siguiendo un enfoque *centrado en datos* o *centrado en procesos* (Baldauf et al., 2007).

El enfoque *centrado en datos* especifica transformaciones en términos de los datos a procesar. Esta especificación aísla enormemente el proceso de transformación de los cambios en los componentes que realmente lo llevan a cabo. Una forma popular de implementar este enfoque es a través de reglas de transformación declarativas. Este tipo de reglas especifican precondiciones y postcondiciones sobre la representación del contexto. Cuando el contexto coincide con la precondición requerida, la regla se dispara para establecer la postcondición especificada. El tipo de agregación que puede hacerse de esta forma está limitado por el poder expresivo del lenguaje de reglas. La OCP (Nieto et al., 2006) usa este enfoque.

Otros trabajos adoptan una implementación imperativa: los elementos que procesan el contexto contienen las instrucciones necesarias para comprobar las condiciones del contexto y llevar a cabo los cambios requeridos. Esta última solución no necesita un intérprete de reglas adicional, como la declarativa, pero en cambio su código es menos legible para el humano. La *iRoom* (Winograd, 2001), la *Context Management Framework* (Korpipaa et al., 2003), y el trabajo de Haya et al. (2006) utilizan esta alternativa.

En el enfoque *orientado a proceso*, la agregación se especifica en términos de los proveedores de contexto. Esto da a los desarrolladores más control sobre cómo tratar con las fuentes de información contextual redundantes, ya que ellos especifican su composición. Sin embargo, este enfoque necesita

adaptación adicional cuando estas fuentes cambian, algo que no ocurre con los enfoques centrados en datos. Este enfoque se usa en el *Context Toolkit* (Dey et al., 2001), los trabajos de Henricksen et al. (2006), Ejigu et al. (2008) y Venturini, Carbó y Molina (2012), y en el proyecto *Hermes* (Buthpitiya et al., 2012). Se destaca aquí el trabajo de Venturini, Carbó y Molina (2012) por su utilización de agentes software, cuestión que se aborda en el presente trabajo. Define el procesamiento del contexto en términos de diagramas de comunicación entre agentes, que especifican los diferentes resultados de los intercambios de elementos del contexto entre los mismos.

2.3.1.4. Distribución de los componentes y organización de la información

Otras características para analizar la infraestructura son la *distribución de los componentes* y la *organización de la información*. La *distribución de los componentes* se refiere a la forma en la que los distintos elementos de un sistema se colocan en dispositivos computacionales, y distingue si la infraestructura usa servidores centrales o no. La *organización de la información* considera las capas de abstracción en las que se divide la información distinguiendo entre organizaciones *estáticas* y *dinámicas*.

El uso de un servidor centralizado libera a los componentes distribuidos de la carga del procesamiento del contexto, y facilita mantener la consistencia de la representación del mismo usando mecanismos genéricos (i.e, no específicos del dominio). Sin embargo, limita la escalabilidad del sistema y lo hace menos tolerante a fallos. *iRoom* (Winograd, 2001), *Hydrogen* (Hofer et al., 2003), the *Context Management Framework* (Korpipaa et al., 2003), y el trabajo de Haya et al. (2006) siguen este enfoque. Cuando no hay un servidor centralizado, los nodos necesitan llevar a cabo procesamiento adicional para mantener la consistencia, pero el sistema en conjunto escala mejor. El *Context Toolkit* (Dey et al., 2001), OCP (Nieto et al., 2006), el proyecto *Hermes* (Buthpitiya et al., 2012), y los trabajos de Henricksen et al. (2006), Ejigu et al. (2008) y Venturini, Carbó y Molina (2012) adoptan este enfoque.

Con respecto a la organización de la información, la organización *estática* en capas significa que las capas de abstracción de la información están predeterminadas por la infraestructura. Esto hace los sistemas más uniformes y facilita la detección de errores. El *Context Toolkit* (Dey et al., 2001), *Hydrogen* (Hofer et al., 2003), OCP (Nieto et al., 2006), y los trabajos de Ejigu et al. (2008) y de Henricksen et al. (2006) usan este enfoque. Por el contrario, la organización en capas *dinámica* implica que las capas de abstracción aparecen en el sistema en tiempo de ejecución. Los sistemas resultantes ganan en flexibilidad para establecer a qué nivel de abstracción trabajar. Sin embargo, esta estructura adaptativa también hace más difícil a los ingenieros detectar errores en el sistema, ya que pueden hacer menos suposiciones acerca del estado actual y estructura del sistema. Este es el enfoque de *iRoom* (Wi-

nograd, 2001), la *Context Management Framework* (Korpipaa et al., 2003), el proyecto *Hermes* (Buthpitiya et al., 2012), y los trabajos de Haya et al. (2006) y de Venturini, Carbó y Molina (2012).

2.3.2. Adaptación y uso del contexto

Usando como base la gestión del contexto, cada una de las arquitecturas facilita el uso de ese contexto para llevar a cabo comportamientos adaptativos. Si bien los comportamientos concretos dependen de la aplicación en desarrollo, y por tanto no se van a tratar en este trabajo, lo que sí se va a tratar es el modo de controlar la elección y ejecución de estos comportamientos cuando sea necesario. Una característica clave de este control es el concepto de **oportunismo**. El oportunismo implica el siguiente proceso: dada una serie de objetivos (tareas de adaptación), evaluar si se pueden llevar a cabo en el contexto actual; en caso de que no se pueda, dejar estos objetivos "suspendidos" y mantenerse observando los cambios de contexto; cuando el contexto sea oportuno, retomar los objetivos suspendidos, y trabajar para completarlos de acuerdo al nuevo contexto existente. Este enfoque abstracto es común en los trabajos a distintos niveles cuando se habla de oportunismo, tal como se explica a continuación. Este tipo de control permite al SSC seleccionar entre una serie de comportamientos según el contexto actual, cada uno de los cuales llevará a cabo una tarea que estará adaptada a las condiciones concretas en las que se ejecuta. A continuación se explican distintas manifestaciones del oportunismo, a distintos niveles de una arquitectura genérica de SSC. La última subsección habla sobre los flujos de trabajo sensibles al contexto, que resultan ser otra forma de control oportunista.

2.3.2.1. Redes oportunistas

Las redes oportunistas son redes donde las conexiones entre nodos son altamente inestables, y las vías de información se establecen usando las conexiones momentáneas con otros nodos (Pelusi et al., 2006). Por este motivo, los nodos intermedios en una conexión extremo a extremo necesitan almacenar temporalmente la información, y esperar a que ciertos nodos se conecten para transmitirla. Además, ciertas aplicaciones están definidas de tal forma que los nodos de destino no se conocen de antemano, sino que se descubren en el proceso de difusión. Esta funcionalidad constituye la forma más básica de oportunismo, y ha sido estudiada en algunas aplicaciones y arquitecturas (Huang et al., 2008). La selección en cada nodo de a qué nodos reenviar la información puede basarse en información contextual adicional. El enfoque más común para esta gestión de contexto es el enfoque centrado en datos (explicado en la sección anterior). Por ejemplo, el proyecto CAMEO (*Context-Aware MiddlEware for Opportunistic mobile social networks*) (Arnaboldi et al., 2011) y Boldrini et al. (2010) proponen un middleware que infiere infor-

mación social de los usuarios para predecir qué nodos se van a conectar más probablemente.

2.3.2.2. Computación oportunista

La computación oportunista es un paso más en estos mecanismos. Consiste en el uso de recursos remotos y posiblemente desconocidos para ejecutar tareas complejas. La idea es similar a las redes oportunistas, pero en este caso las peticiones remotas de información se refieren a elementos abstractos que deben ser resueltos. La implementación exacta para estas peticiones se deja a los nodos de destino (Conti et al., 2010). Este tipo de comportamiento necesita la definición de modelos abstractos de información y definiciones abstractas de sensores. De este modo, la misma información abstracta puede implementarse por distintos procesadores concretos en distintas circunstancias. Este enfoque transforma la red oportunista en una nube de recursos abstractos utilizables por cualquier nodo de la red. Un ejemplo de estos trabajos es el de Kurz y Ferscha (2010), que propone una colección de abstracciones de sensores y un mecanismo de autodescripción para ellos. Esta arquitectura puede extenderse replicando los procesos descritos en múltiples capas de abstracción.

2.3.2.3. Fusión de información oportunista

La fusión de información oportunista se define en el siguiente nivel de abstracción. Este tipo de mecanismo consiste en decidir las fuentes exactas de información a usar para crear información a un mayor nivel de abstracción. Esta decisión se basa en el contexto actual, y es oportunista, es decir, basada en ciertas métricas tomadas de los recursos disponibles, preferencias del usuario, actividades actuales, localización, etc. Usando este principio, una aplicación no sólo es capaz de obtener información de diferentes fuentes abstractas, sino incluso de cambiar entre distintas fuentes dependiendo de la situación. Challa et al. (2005) proponen conceptos y alternativas para este aspecto, tales como la modificación de reglas de fusión como respuesta a cambios de contexto.

2.3.2.4. Planificación oportunista

La planificación oportunista corresponde al nivel de abstracción más alto. Consiste en definir actividades complejas a nivel de aplicación, que son ejecutadas de forma sensible al contexto y oportunista. En este nivel, la selección de tareas de fusión es transparente. El objetivo es tener mecanismos que faciliten la detección de situaciones en las cuales la ejecución de ciertas actividades es conveniente por alguna razón. Por ejemplo, esto puede ser útil para recomendar posibles regalos para el cumpleaños de un amigo cuan-

do la fecha está llegando y el usuario está cerca de una tienda de regalos. En esta línea, el proyecto *OPPORTUNITY* (Roggen et al., 2009) presenta algunos sistemas de reconocimiento de actividades y de contexto con comportamientos oportunistas. También introduce una arquitectura enfocada al mecanismo de fusión de información, pero no especifica los detalles en el modelado y la gestión de la distribución del contexto.

Los tipos de oportunismo anteriores no representan esfuerzos aislados. Para tener acceso a uno de ellos, normalmente es necesario tener implementados los niveles inferiores. Estos tipos pueden combinarse de diferentes formas, lo que produce diferentes arquitecturas.

2.3.2.5. Flujos de trabajo sensibles al contexto

En esta sección se discuten algunas características del soporte de flujos de trabajo sensibles al contexto en la literatura

Con respecto a la gestión de flujos de trabajo, la mayoría de las soluciones separan claramente el control de la ejecución de los mismos y la gestión del contexto. Ranganathan y McFaddin (2004), uFlow (Han et al., 2006) y CAWE (*Context-Aware Workflow Execution*) (Ardissono et al., 2007) son ejemplos de esto. Los dos primeros proponen envolver el sistema de gestión del contexto y usarlo para comprobar condiciones en un entorno de ejecución de flujos de trabajo, mientras que el tercero envuelve la gestión del flujo de trabajo como otro proveedor/consumidor de contexto más en una arquitectura sensible al contexto. El enfoque de Ranganathan y McFaddin (2004) usa el CSC (Componente Sensible al Contexto, *Context-Aware Component*) para escoger una definición de flujo de trabajo adecuada, y entonces procede a su ejecución. Sin embargo, no considera el cambio de circunstancias después de que la definición se haya escogido. Por el contrario, las otras dos alternativas trabajan con una definición abstracta, cuyas acciones se instancian en tiempo de ejecución dependiendo de las condiciones actuales del contexto.

Para las definiciones de los flujos de trabajo, hay múltiples alternativas. uFlow (Han et al., 2006) y CAWE (Ardissono et al., 2007) implementan alternativas propias. Otras propuestas se decantan por aproximaciones ya existentes, como Ranganathan y McFaddin (2004), que proponen usar BPEL (*Business Process Execution Language*), un lenguaje estándar de definición de procesos de negocio. La ventaja de las alternativas orientadas al contexto sobre el uso de propuestas generales es que incluyen conceptos dependientes del contexto, lo que permite definir las condiciones y acciones de los flujos de trabajo en términos de información contextual. Sin embargo, la segunda alternativa utiliza lenguajes o frameworks bien conocidos y establecidos, lo que facilita su aprendizaje e incrementa las posibilidades de disponer de infraestructura y herramientas desarrolladas por terceros.

2.3.3. Uso de agentes en sistemas sensibles al contexto

El uso del paradigma de agentes está también extendido en la literatura IAm, ya que este tipo de sistemas ofrece las capacidades que demandan casos de estudio comunes en el área (Tapia, Abraham, Corchado y Alonso, 2010; Sánchez-Pi, Mangina, Carbó y Molina, 2010).

El concepto de agente permite definir comportamientos muy complejos a un nivel de abstracción muy alto. Una de las capacidades más destacables de los sistemas multi-agente es la de colaborar y negociar. El trabajo de Venturini, Carbó y Molina (2012) muestra cómo el análisis de las interacciones entre agentes puede utilizarse de forma genérica para comparar ciertas propiedades en distintos sistemas multi-agente. En Rodríguez et al. (2010) se hace uso de colaboración para llevar a cabo fusión de la información procedente de múltiples cámaras con el objetivo de detectar objetos. En un trabajo posterior de Venturini, Carbo y Molina (2013) extienden el modelo colaborativo incluyendo reputación para discriminar qué agentes ofrecen mejor información en los distintos casos. Esta capacidad es muy interesante en cualquier SSC para disponer de criterios extra para llevar a cabo la fusión de distintas fuentes de información que pueden ser parcialmente redundantes. La autonomía y capacidad de aprendizaje de los agentes también se suele utilizar para implementar gestión de diálogo. El trabajo de Griol, Carbó y Molina (2012) muestra técnicas de aprendizaje para mejorar el diálogo con el usuario en un entorno inteligente.

La colaboración entre agentes puede también definirse en forma de flujos de trabajo, incluyendo también las actividades propias de los usuarios, que también participan como actores. Un ejemplo notable de detección de actividades es Aiello et al. (2011), que describe una infraestructura para desarrollar redes de sensores inalámbricos usando agentes móviles. El trabajo no proporciona soporte para definir y monitorizar flujos de trabajo. En su lugar, considera la detección de actividades por medio de sensores corporales. La definición de estas actividades es principalmente conseguida por medio de aprendizaje automático asistido, ya que no hay una definición explícita de las mismas. Esto ofrece una gran flexibilidad para monitorizar actividades y cambiar su definición en tiempo de ejecución. La desventaja es la complejidad limitada de las actividades aprendidas teniendo en cuenta los tipos de acciones y número de participantes. Otro ejemplo es CAKE (*Collaborative Agent-based Knowledge Engine*) (Bergmann, 2007), en el cual los agentes se organizan usando definiciones abstractas de flujos de trabajo, dependiendo de la situación obtenida de un razonador basado en casos. Sin embargo, la arquitectura no ofrece soporte especifico para facilitar fusión de información para identificar el caso más adecuado. En su lugar, se basa en la implementación de los agentes para determinar la situación y especificársela al razonador.

Por otro lado, también existe el uso de simulación basada en agentes

como medio de validar aplicaciones IAm, en concreto el trabajo de Campillo-Sanchez, Serrano y Botía (2013) muestra cómo aplicar la plataforma UbikSim para aplicaciones móviles en entornos inteligentes. En la presente tesis se utiliza esta misma plataforma para generar entornos de simulación virtual.

2.4. Conclusiones

El estudio de los trabajos más relevantes dentro de la IAm, y en concreto de los SSC, ha permitido observar ciertas características a tener en cuenta en el diseño de este tipo de sistemas. En concreto, se han observado algunas ventajas de adoptar una arquitectura basada en el modelo de pizarra, que reúne las ventajas de la obtención y procesamiento del contexto "orientada a datos", y además proporciona ventajas para la organización dinámica de los componentes del sistema. Sin embargo, muchos de los trabajos optan por incluir un servidor central que gestiona la información, y se ha valorado que esto puede ser problemático en entornos muy cambiantes, impidiendo que ciertos componentes distribuidos funcionen si dejan de tener acceso al servidor. Aunque hay otras arquitecturas que pueden trabajar temporalmente en ausencia del servidor central, se quiere explorar una alternativa a las estudiadas que no dependa de un servidor único en ningún momento.

Por otra parte, las arquitecturas se enfocan en la gestión de la información del contexto, pero no tratan la forma de hacer uso del mismo. Pocos enfoques describen cómo sus componentes se comportan cuando cambia la configuración del sistema, es decir, cómo se adapta realmente a los cambios en sus partes. Sin embargo, la adaptación al contexto es una parte fundamental de los sistemas IAm. Por ello, este trabajo pretende incluir patrones para facilitar modificaciones de comportamiento del sistema de acuerdo a cambios en el contexto, para de este modo poder llevar a cabo los comportamientos adaptativos.

Por último, muchas de las arquitecturas que se presentan están rígidamente estructuradas para controlar el acceso, difusión y procesamiento de la información. Esta característica, pese a favorecer la rigurosidad y la operatividad de las arquitecturas, implica un esfuerzo mayor para implementar prototipos simples. A este respecto, este trabajo plantea una arquitectura que no imponga más que restricciones organizativas básicas a la hora de implementar aplicaciones. El objetivo es poder desarrollar prototipos funcionales sin necesidad de especificar una estructura compleja detalladamente, de tal forma que después puedan ser mejorados si esta estructura es necesaria. La arquitectura además debe incluir en su descripción aquellas cuestiones que tienen que ver con la forma de abordar un desarrollo con la arquitectura: cómo se estructuran los proyectos, cómo se prueban y validan, etc.

En el próximo capítulo se describen cuáles han sido las decisiones principales para abordar estas cuestiones. Los capítulos posteriores desarrollarán

esta arquitectura elaborada según esas pautas en cada una de sus diferentes capas.

Capítulo 3

Enfoque del problema

Las únicas cosas que el mago necesita absolutamente para practicar la magia son la voluntad para imponer sus deseos a la realidad, el conocimiento de las necesarias Esferas, y el paradigma a través del cual enfocar ese deseo y convertirlo en realidad.

Fragmento de la definición de Paradigma.
Mago: la Ascensión, capítulo 4.

Resumen: En este capítulo se hace un análisis, a la luz del estudio de las características de los sistemas IAm en el capítulo anterior, de cuáles son los requisitos que debe cumplir la arquitectura objetivo, y qué decisiones se toman de partida para enfocar el diseño posterior.

3.1. Identificación de requisitos

El capítulo anterior hace un análisis de algunas de las arquitecturas más representativas aplicadas al desarrollo de SSC. Éstas responden a una serie de requisitos comunes al tipo de escenarios que se contemplan en el campo de la IAm. Estos requisitos incluyen la naturaleza "ambiental" de las aplicaciones, es decir, la disponibilidad flexible de sus recursos y servicios y la intromisión mínima en el desarrollo de las actividades de los usuarios. También incluyen su naturaleza "adaptativa", que implica que éstas sean capaces de deducir qué ocurre en su ámbito y actuar en consecuencia. Estos requisitos generales se traducen a otros más particulares que establecen las características básicas de aquello de lo que debe ser capaz un sistema desarrollado en base a estas

arquitecturas. La procedencia de estos requisitos se especifica en los trabajos citados en la siguiente lista:

a. Ser instalable en diversos dispositivos, incluso en aquellos con bajas prestaciones. Esto es necesario para que los sistemas sean capaces de aprovechar todos los recursos que tengan a su alcance para llevar a cabo las tareas (Abowd et al., 1999).

b. Encontrar nuevos recursos disponibles automáticamente y ser capaz de utilizarlos. De este modo se reducen los esfuerzos de configuración por parte de los usuarios (York y Pendharkar, 2004).

c. Funcionar en un entorno de componentes distribuidos en una red. Gracias a esto, el sistema es más robusto, porque no depende únicamente de una máquina (Pelusi et al., 2006).

d. Actualizar la representación del contexto en tiempo real conforme a los cambios detectados, y permitir llevar a cabo razonamiento sobre el mismo. Esto permite que las aplicaciones puedan conocer qué está pasando de una forma abstracta para decidir qué hacer (Haya et al., 2006).

e. Proporcionar mecanismos de adaptación para las aplicaciones (reconfiguración, detección de actividades, personalización, etc.). Esto es, mecanismos que faciliten que las aplicaciones adapten su comportamiento de manera dinámica ante los cambios (Kurz y Ferscha, 2010).

f. Permitir ampliar su tamaño de forma sencilla, para evitar problemas típicos de escalabilidad y mantener la robustez. Este requisito es básico de cualquier tipo de arquitectura distribuida.

Aparte de estos requisitos mínimos, hay otros adicionales que han sido tenidos en cuenta para este trabajo. Cada uno de ellos responde a unos motivos que se explican a continuación:

g. Permitir modificar la funcionalidad sin necesidad de parar el sistema. No todas las arquitecturas estudiadas tratan este tema de forma explícita, pero esta característica resulta interesante para reforzar el requisito de no "interrumpir" a los usuarios. De este modo, no es necesario detener las aplicaciones en ejecución para instalar otras nuevas (Remagnino et al., 2005).

h. Permitir el procesamiento de información del contexto de cualquier origen. Muchas de las arquitecturas estudiadas establecen una serie de capas de abstracción de información "prediseñadas". Aunque esto resulta útil para estructurar aplicaciones, supone una dificultad cuando

se quieren hacer prototipos simples. Si esta división en capas no es forzada, el desarrollador puede implementarla igualmente si lo considera necesario (Buthpitiya et al., 2012).

i. En el modelado del contexto, no forzar el uso de motores de razonamiento externos, y permitir el modelado simple orientado a objetos. Aunque las ventajas expresivas de otros lenguajes de modelado de información, como las ontologías, son indiscutibles, ninguno de los sistemas estudiados hace una definición de su contexto ni un procesamiento posterior tan complejo que haga necesario el uso de esta potencia. Si bien es cierto que esta necesidad puede surgir conforme evoluciona un sistema, ésta puede solucionarse de otras maneras. Por ejemplo, en el trabajo de Kalyanpur et al. (2004), se propone una manera de convertir ontologías OWL en interfaces y clases Java.

Los requisitos anteriores cubren los objetivos tecnológicos identificados en el primer capítulo. A continuación se identifican una serie de requisitos relacionados con el modelo de desarrollo asociado a la arquitectura objetivo. El motivo de incluir estos requisitos es favorecer el uso de buenas prácticas de ingeniería del software, y permitir llevar a cabo prototipos de sistemas mediante un método consistente. El desarrollo de un sistema con la arquitectura propuesta en esta tesis, FAERIE, debe seguir las siguientes pautas:

I. Proporcionar un conjunto de plantillas configurables para la creación de nuevos proyectos. Estas plantillas especifican la estructura de los ficheros y los módulos de un proyecto.

II. Favorecer el desarrollo con Integración Continua, es decir, integrar la infraestructura en un entorno que ejecute test unitarios, y sea capaz de construir elementos desplegables con distintas configuraciones de forma automática.

III. Facilitar la usabilidad para el desarrollador adaptando su uso a IDE (*Integrated Development Environment*, Entorno de Desarrollo Integrado) existentes como Eclipse o Netbeans.

IV. Realizar integración automática con un entorno de simulación 3D para la validación de las aplicaciones.

V. Facilitar el despliegue e instalación para su uso.

3.2. Decisiones de diseño y desarrollo

Dados los requisitos enunciados en la sección anterior, ahora es necesario especificar de qué manera implementarlos en la arquitectura resultante.

Para ello se van a explicar una serie de decisiones de partida que determinan parte de la estructura y comportamiento de la arquitectura que se va a desarrollar en los próximos capítulos. Los requisitos a., b., c. y g. pueden conseguirse mediante una infraestructura basada en componentes, como se describe a continuación. Los requisitos d., e., f., h. e i. se consiguen mediante una arquitectura basada en el modelo de pizarra. Los requisitos de desarrollo indicados se consiguen mediante el uso de una herramienta para la gestión de proyectos del tipo de Apache Maven (Massol y O'Brien, 2005), lo que facilita lograr el requisito II. como se explicará más adelante. A su vez, esta herramienta también incluye soporte para los requisitos I. y V., y además permite su adaptación para cumplir III. Mediante el uso del entorno de simulación UbikSim (Campuzano et al., 2011), se consigue cumplir el requisito IV.

3.2.1. Infraestructura basada en componentes

En primer lugar, se va a optar por construir una infraestructura basada en componentes dinámicos. Esto implica que un sistema constará de varios módulos con las siguientes partes: un conjunto de interfaces proporcionadas, un conjunto de interfaces requeridas, y un conjunto de propiedades clave-valor. En este tipo de infraestructuras, que se explicarán en detalle en el Capítulo 5, se llevan a cabo dos tareas principales: gestionar el ciclo de vida de los componentes, es decir, controlar el despliegue o repliegue de los mismos; y gestionar el enlace de interfaces entre componentes, es decir, buscar automáticamente aquellos componentes que proporcionan interfaces que otros necesitan y asociarlos.

El principal motivo de esta elección es que este tipo de infraestructura posee unas características fácilmente adaptables a lo que se desea conseguir, como se explicará en el capítulo correspondiente. Concretamente, se hará uso de la plataforma OSGi (*Open Services Gateway Initiative*) (Rellermeyer et al., 2007), la cual ya tiene soporte para distintas plataformas y sistemas operativos, incluyendo algunos protocolos comunes como UPnP (*Universal Plug and Play*) (Schneider et al., 2001). Sin embargo, la definición de la infraestructura va a abstraerse del uso concreto de OSGi, para admitir cualquier otra infraestructura que pueda ofrecer características parecidas, como CORBA (Pope, 1998) o RMI (Plasil y Stal, 1998), las cuales ya han sido usadas en otros trabajos.

El objetivo de OSGi es diseñar plataformas compatibles que puedan proporcionar múltiples servicios. Su concepción está orientada a su aplicación en redes domésticas y en domótica o informatización del hogar. Aunque OSGi define su propia arquitectura, es compatible con UPnP. Además, define una serie de API básicas para el desarrollo de servicios, como los de registro, servidor HTTP (*HyperText Transfer Protocol*) y el DAS (*Device Access Specification*), que permite descubrir los dispositivos y servicios ofrecidos por éstos.

La principal alternativa al uso de este tipo de infraestructura es una basada en organizaciones de agentes (Chen et al., 2003). Este tipo sería capaz de ofrecer las capacidades dinámicas y distribuidas que requieren los sistemas IAm. Sin embargo, se ha considerado que los procesos que se llevan a cabo en los sistemas IAm a bajo nivel consisten principalmente en tareas de fusión de información y adaptación que no requieren de la complejidad de modelado de un agente software. Por el contrario, se postula el uso de esta abstracción en capas más elevadas de la arquitectura.

3.2.2. Modelo de pizarras federadas

En segundo lugar, de entre las opciones de distribución y organización de los componentes mencionadas en el estado del arte, se va a llevar a cabo una arquitectura basada en un modelo de pizarras federadas. Esto quiere decir que, dentro del sistema de componentes distribuidos, va a existir un conjunto de elementos centrales que van ser los contenedores donde se almacenará el modelo del contexto. Cada una de estas pizarras mantendrá una visión parcial del contexto del sistema, pero estarán comunicadas de tal manera que podrán transmitirse información cuando sea necesario, para satisfacer las peticiones de sus componentes vecinos. De este modo, al no haber una única pizarra centralizada, se permitirá la redundancia de la información, mejorando la robustez y la escalabilidad. Por otra parte, el mantener varias representaciones del contexto tiene la desventaja de que es necesario implementar mecanismos que aseguren la consistencia de la información. El Capítulo 6 trata este aspecto en mayor detalle.

La principal alternativa a esta decisión es el uso de una pizarra virtual (Nieto et al., 2006). En este caso, no existe un contenedor del contexto físico, sino virtual. De este modo, cuando un componente accede a él, en realidad está accediendo directamente a los proveedores de la información. Esto tiene la ventaja de no requerir ningún tipo de espacio de almacenaje extra. Sin embargo, esta versión no permite implementar métodos de eficiencia, como la creación de una caché, o de robustez, como la replicación de la información en múltiples pizarras.

3.2.3. Gestión de proyectos e integración continua

En tercer lugar, se va gestionar la creación de proyectos mediante una herramienta que facilite el desarrollo mediante integración continua, como es el ejemplo de Apache Maven. Maven construye distribuciones de proyectos software formados por un conjunto de módulos interdependientes. Para ello sigue un ciclo en el que lleva a cabo una serie de procesos en un determinado orden:

1. valida los proyectos y obtiene todas las dependencias que necesiten;

2. compila los proyectos;
3. ejecuta pruebas unitarias;
4. empaqueta los módulos;
5. ejecuta pruebas de integración entre módulos;
6. comprueba que los paquetes cumplen ciertos criterios especificados;
7. instala los paquetes en el repositorio local;
8. y despliega los paquetes en el repositorio remoto.

Este ciclo de procesos permite tener control sobre cada fase del desarrollo de los proyectos. Además, el uso de repositorios locales y remotos permite gestionar fácilmente el uso de bibliotecas externas.

Una de las utilidades más interesantes de esta herramienta es la definición de arquetipos y de perfiles de distribución. Un arquetipo es una definición abstracta de la estructura de un tipo de proyecto software. Haciendo uso de arquetipos, se puede generar automáticamente la estructura de ficheros básica, y algunos ficheros fuente a partir de los cuales continuar el desarrollo. Uno de los resultados del presente trabajo es un conjunto de arquetipos para crear proyectos de aplicaciones IAm. Un perfil de distribución es una definición de la estructura de los elementos ejecutables generados. Haciendo uso de estos perfiles se pueden especificar distribuciones para distintos tipos de plataformas. Este trabajo también hace uso de los perfiles, lo cual resulta muy útil para sistemas IAm, donde hay una gran heterogeneidad entre las plataformas de destino.

La decisión de hacer uso de la herramienta Apache Maven puede reconsiderarse en el futuro, para incluir otras que también favorezcan el desarrollo mediante integración continua. En todo caso, otra alternativa a su uso sería simplemente proporcionar scripts o emplear entornos de desarrollo que faciliten la gestión de las pruebas unitarias y el despliegue de los módulos.

Como objetivo a largo plazo de la arquitectura se considera además establecer un marco de desarrollo que favorezca la reutilización del código. Para ello puede ser necesario definir algún tipo de vocabulario para unificar conceptos y funcionalidades, eliminando las posibles ambigüedades en este aspecto, como se propone en Vizcaíno et al. (2012).

Alternativas más simples a este modelo de gestión del desarrollo de los proyectos son aquellas que se limitan a la gestión de versiones de software. Sin embargo, éstas alternativas no tienen en consideración las implicaciones de tener múltiples versiones de distintos componentes ejecutándose a la vez en el sistema, y de cómo afectan las dependencias entre los distintos módulos a la hora del desarrollo. En este sentido, las alternativas que sólo consideran la versión del código no son lo suficientemente potentes.

3.2.4. Pruebas y validación con escenarios virtuales

Por último, se va a integrar dentro del ciclo de desarrollo una herramienta de simulación de escenarios 3D que permita desplegar dispositivos virtuales en el mismo. Esto permitirá hacer validaciones de las aplicaciones previas a su despliegue real. Esto tiene la ventaja del coste considerablemente reducido de subsanar los errores que se encuentren en fases tempranas del desarrollo en comparación con las posteriores, ya que ello no requerirá un despliegue real, con dispositivos y personas reales. Haciendo uso de este tipo de software pueden ejecutarse pruebas en lotes, y reproducir situaciones exactas.

El software elegido para estas pruebas es UbikSim (Serrano y Botía, 2013), una herramienta de simulación que ofrece las facilidades indicadas. FAERIE incluirá un adaptador para integrar los sensores virtuales declarados en el entorno UbikSim, de tal manera que su uso sea transparente. El Capítulo 10 explica esta integración en mayor detalle.

La principal alternativa a esta decisión es la habilitación de entornos de pruebas reales con sujetos reales. Esta es la solución utilizada en los laboratorios vivientes o "*living labs*" (Bergvall-Kareborn et al., 2009), donde se despliegan los sistemas en entornos físicos haciendo uso de dispositivos físicos. Esto tiene la obvia desventaja del enorme coste para la creación de prototipos, y de ahí que haya sido descartada.

3.3. Conclusiones

Los requisitos identificados en este capítulo, comunes a multitud de aplicaciones IAm, deberán tenerse en cuenta a la hora de definir la arquitectura. Estos requisitos básicos abordan las capacidades mencionadas del sistema con respecto a su funcionamiento. Además, se agregan otros que simplifican el desarrollo rápido de aplicaciones. Sobre estos se incluyen requisitos que tienen que ver con proporcionar mecanismos de creación, pruebas y validación de los sistemas.

Después de la identificación de requisitos se han adoptado una serie de decisiones para orientar la arquitectura. La primera, estar basada en componentes, debido a la existencia de múltiples plataformas de este tipo que proporcionan características deseables para el desarrollo propuesto. La segunda, basar la arquitectura en un modelo de pizarras distribuidas. El modelo típico de arquitectura basada en pizarra, que ha sido estudiado en el capítulo anterior, hace uso de un servidor central. La arquitectura FAERIE pretende estudiar una versión distribuida, mediante la federación de múltiples pizarras localizadas en nodos distribuidos. La tercera, orientar la infraestructura para hacer uso de herramientas que faciliten una buena gestión desde el punto de vista de la Ingeniería del Software. Finalmente, la cuarta persigue reducir el coste de las pruebas y validaciones de los sistemas haciendo uso de la

simulación en escenarios virtuales.

En resumen, la infraestructura FAERIE constará de una serie de componentes que ofrezcan los servicios básicos y comunes de las aplicaciones IAm. Mediante el uso de los arquetipos Maven anteriormente citados, podrá generarse un proyecto que haga uso de estos componentes, el cual estará formado por distintos módulos. En este proyecto podrán llevarse a cabo pruebas a distintos niveles (unitarias, de integración, simulaciones de sistema y pruebas de aceptación de requisitos), y ser desplegado en distintas plataformas. Una vez en ejecución, FAERIE mantendrá la comunicación entre las distintas representaciones del contexto, y permitirá arrancar dinámicamente nuevos módulos que agreguen funcionalidad a las aplicaciones.

En el próximo capítulo se describe en mayor detalle la estructura y comportamiento de la arquitectura FAERIE. Los distintos aspectos son motivados en base a los requisitos descritos en este capítulo.

Capítulo 4

La arquitectura FAERIE

No se ofusque con este terror tecnológico que ha construido. La posibilidad de destruir un planeta es algo insignificante comparado con el poder de la Fuerza.

Darth Vader a un general Imperial.
Star Wars episodio IV.

Resumen: En este capítulo se ofrece una visión general de la arquitectura y de sus distintas partes desde dos perspectivas diferentes: la organización en capas de la funcionalidad ofrecida por la infraestructura, y la organización y relaciones entre los componentes e interfaces que forman un SSC.

4.1. Introducción

Teniendo en cuenta el enfoque escogido en el capítulo anterior, en este capítulo se presenta una introducción a la arquitectura de FAERIE para cumplir los requisitos identificados. Para describir la arquitectura se va hacer uso de dos perspectivas: la organización funcional (Sección 4.2) y la organización de componentes e interfaces (Sección 4.3). La primera perspectiva ofrece una visión dividida en capas de la arquitectura, donde se especifican un conjunto de servicios o funcionalidades que cada capa ofrece a la capa superior. La segunda perspectiva ofrece una visión de los tipos de componentes fundamentales en la arquitectura, que son aquellos que tienen la responsabilidad de ofrecer las funcionalidades anteriormente descritas. En cada uno de los aspectos estudiados, se especificará qué funciones tiene y qué requisitos persigue, según su numeración en la sección 3.1.

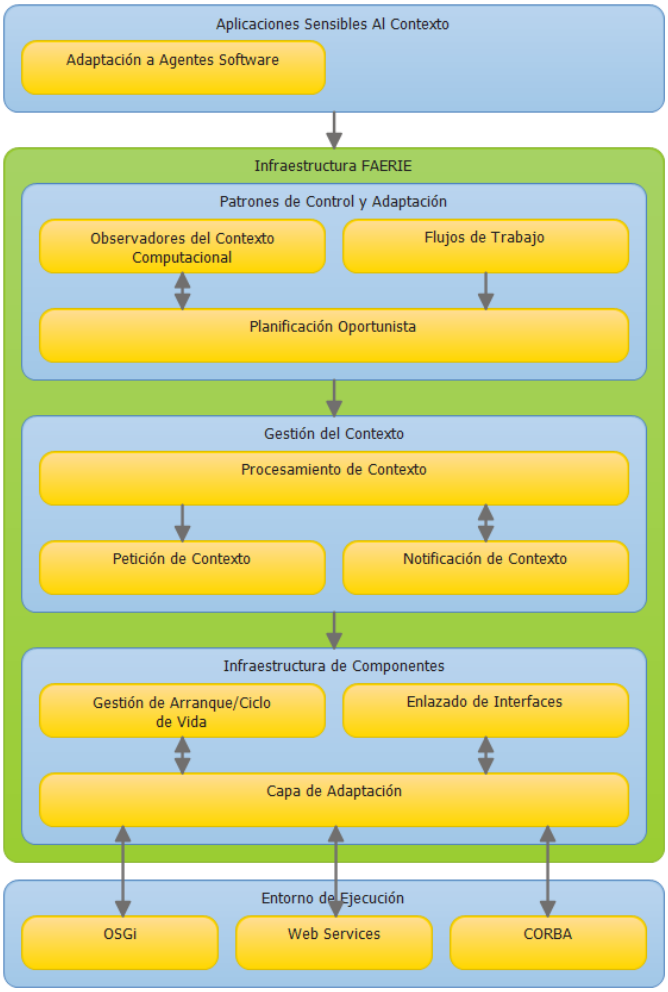

Figura 4.1: Organización en capas de la infraestructura FAERIE.

4.2. Organización funcional

La Figura 4.1 muestra las capas de funcionalidad de la infraestructura FAERIE. Como puede observarse, la infraestructura se divide en tres capas, y a su vez se apoya en un entorno de ejecución determinado. Por encima de la infraestructura se sitúan las aplicaciones sensibles al contexto desarrolladas a partir de ella. Hay que subrayar que estas aplicaciones no están limitadas en el uso de las distintas funcionalidades, y además, pueden organizar el acceso a los distintos niveles de información abstracta de diversas formas, tal y como indica el requisito h.

4.2.1. Capa de entorno de ejecución

Esta capa representa el entorno computacional, normalmente desarrollado por terceros, que va a ejecutar tanto los componentes de la infraestructura FAERIE como los de las aplicaciones que en última instancia hacen uso de la misma. En esta capa se han considerado distintos tipos de entornos, como por ejemplo OSGi (Rellermeyer et al., 2007), Web Services (Alonso et al., 2004) o CORBA (Pope, 1998). Su objetivo en esta capa es proporcionar una serie de servicios básicos de conexión y ciclo de vida. Además, así se cumple con el requisito a. (instalable en dispositivos diversos), ya que estos entornos son muy populares y tienen implementaciones para múltiples plataformas. Estas tecnologías son recubiertas por la siguiente capa para hacer el resto de la infraestructura independiente de ellas, de tal modo que en el futuro se pueden considerar otras implementaciones.

4.2.2. Capa de infraestructura de componentes

Esta capa proporciona los servicios necesarios para el desarrollo de los componentes de aplicaciones sensibles al contexto, y su adaptación al entorno de ejecución correspondiente. Sus objetivos principales son: gestionar el lanzamiento dinámico de nuevos módulos en el sistema, cumpliendo el requisito g.; e interconectar unos módulos con otros en base a una serie de interfaces y propiedades, incluyendo módulos que se encuentren distribuidos en una red, cumpliendo los requisitos b. y c.

Los servicios que constituyen esta capa ofrecen el arranque y la comunicación básica entre componentes. Se describirán con mayor detalle en el Capítulo 5. Consta a su vez de varias capas:

Capa de Adaptación Adapta la funcionalidad ofrecida por la plataforma concreta de ejecución, por ejemplo a OSGi, RMI o CORBA. Ofrece al resto de las capas una visión unificada de la infraestructura subyacente. De esta forma, el resto de componentes siempre pueden trabajar asumiendo los mismos componentes e interfaces en las capas inferiores, con independencia de la capa de entorno de ejecución que se esté usando realmente.

Gestión de Arranque/Ciclo de Vida Gestiona el arranque de nuevos componentes en la plataforma, controlando las operaciones que éstos llevan a cabo durante el proceso. Este arranque puede iniciarse bajo demanda del usuario o de forma automática al detectarse eventos por parte de la infraestructura o de las aplicaciones.

Enlazado de Interfaces Se encarga de poner en contacto elementos por medio de sus interfaces públicas. Además, si alguna interfaz se declara como remota, también se hace accesible a otros nodos dentro de una red.

4.2.3. Capa de gestión del contexto

Esta capa se encarga de generar la abstracción del contexto del sistema y dar acceso a la misma. Coordina automáticamente los componentes para establecer los flujos de agregación e interpretación de información que actualizan la representación del contexto. De este modo cumple el requisito d.

Los servicios incluidos en esta capa ofrecen la posibilidad de solicitar cambios en el contexto, suscribirse a los mismos, o declarar transformaciones de contexto. El Capítulo 6 presenta esta funcionalidad con mayor detalle. Esta funcionalidad se ha estructurado en las siguientes partes:

Petición de Contexto Permite hacer una petición de información de contexto. Esta petición genera una coordinación automática de procesos que culmina en la obtención de la información deseada.

Notificación de Contexto Permite suscribirse a ciertos tipos de cambios en el contexto. Funciona como el patrón Observador (Gamma et al., 1995), ejecutando un determinado procedimiento al detectarse cada cambio.

Procesamiento de Contexto Permite especificar un *observador del contexto* que transforme un tipo de información de contexto en otra.

4.2.4. Capa de patrones de control y adaptación

Sobre la capa anterior, esta capa proporciona una serie de patrones repetibles para desarrollar comportamientos adaptativos. En concreto, ofrece la posibilidad de definir modos de comportamiento en base a condiciones en el contexto. Esta capa cumple el requisito e.

La funcionalidad para describir comportamientos adaptativos permite también ofrecer patrones adicionales para cambiar el modo en el que se procesa la información, y especificar comportamientos en función del estado de flujos de trabajo complejos. El Capítulo 7 explica con detalle estos patrones. La funcionalidad de la capa se organiza en las siguientes partes:

Planificación Oportunista Permite establecer cambios automáticos de comportamiento según ciertas condiciones del contexto. Además, ofrece una serie de comportamientos genéricos que se pueden reutilizar.

Observadores del Contexto Computacional Permite declarar observadores que cambian el modo en el que se procesa el contexto según las condiciones del propio contexto. Un ejemplo es cambiar los criterios de agregación de información para escoger sólo las fuentes más fiables o más baratas según las circunstancias concretas.

Flujos de Trabajo Permite declarar flujos de trabajo para establecer actividades a realizar o monitorizar. De este modo, el comportamiento cambia según el "estado" del flujo de trabajo.

4.2.5. Capa de aplicaciones sensibles al contexto

En esta capa es donde se sitúan las aplicaciones sensibles al contexto que se desarrollan a partir de FAERIE. En ciertos casos, estas aplicaciones harán uso del paradigma multi-agente. Este enfoque es muy popular en los sistemas IAm, ya que ofrece un nivel de abstracción lo bastante alto como para definir comportamientos muy complejos (Venturini, Carbó y Molina, 2012). Además, muchos mecanismos desarrollados para sistemas multi-agente resultan útiles en este tipo de sistemas, por ejemplo, la especificación y tratamiento de secuencias de diálogo, tanto con usuarios como con otros agentes.

4.3. Organización de componentes e interfaces

La Figura 4.2 representa la estructura de un SSC (*Context-Aware System*) según FAERIE. Un SSC S se concibe como un conjunto de *entornos* (*environments*) interconectados $\varepsilon_1, \ldots, \varepsilon_n$, donde cada uno de los entornos ε_i comprende:

- un *entorno físico* (*physical environment*) p_i, que contiene un único **dispositivo principal** (*main device*) d_{i0} y un conjunto de **dispositivos periféricos** (*peripheral devices*) $d_{i1}, \ldots, d_{im}$ conectados físicamente con d_{i0}. El tipo P se define como el conjunto de todas las posibles combinaciones de estados de dispositivos en un entorno físico. La *localización* y *cobertura* de p_i están determinadas por la localización, tipo y rango (*range*) de los dispositivos con capacidades de actuación o percepción en él, y pueden cambiar a lo largo del tiempo. Por tanto, múltiples entornos físicos de distintos entornos pueden solaparse.

- un *entorno computacional* (*computational environment*) c_i conectado en una red (denominado **nodo FAERIE**), que se representa mediante la infraestructura software ejecutándose en d_{i0}. Contiene los componentes que implementan la lógica de las aplicaciones IAm. Concretamente, un *contenedor de contexto* (*context container*) κ_i que mantiene el *modelo de contexto* bajo la forma de un conjunto de elementos de contexto, y un conjunto de *observadores del contexto* (*context observers*) $o_{i1}, \ldots, o_{ip}$ que manipulan este modelo. El tipo K se define como el conjunto de todas las posibles combinaciones de estados de elementos de contexto en un contenedor del contexto. Estos elementos se describirán en mayor detalle en el Capítulo 6.

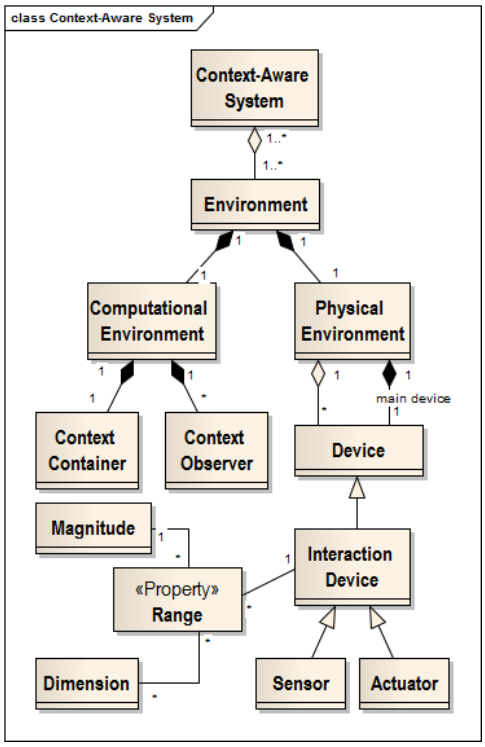

Figura 4.2: Concepción de SSC en FAERIE.

- un conjunto de *entornos conocidos* $\varepsilon_{i1}, \ldots, \varepsilon_{iq}$ tales que cada nodo $c_{i1}, \ldots, c_{iq}$ está en la misma red que c_i, conectados mediante un tipo de componente denominado **observadores del contexto remoto** (*remote context observers*).

Un ejemplo de conexión entre nodos se modela en la Figura 4.3. Muestra tres dispositivos principales: un "smartphone", un ordenador de escritorio y un portátil. Hay algunos dispositivos periféricos (sensores y actuadores), aplicaciones y ficheros. Cada dispositivo principal ejecuta su nodo, que contiene dos tipos principales de componentes: un contenedor del contexto y múltiples observadores. Algunos de estos observadores manejan los dispositivos periféricos.

A continuación se explican algunos de los elementos mencionados en la descripción anterior.

Dispositivos principales Son los dispositivos que ejecutan los componentes de la infraestructura. Sólo hay uno de ellos en cada entorno inteligente, y normalmente poseen la mayor capacidad de procesamiento.

Dispositivos periféricos Estos dispositivos incluyen los sensores y actua-

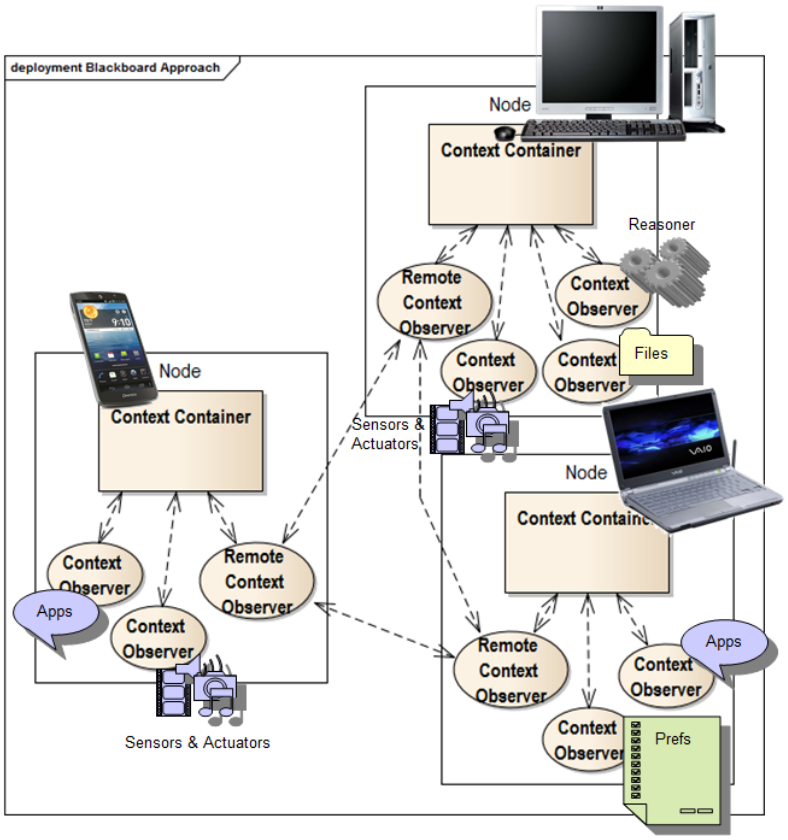

Figura 4.3: Ejemplo de sistema FAERIE.

dores desplegados en el espacio, al igual que otros elementos que pueden proporcionar servicios adicionales (p.ej. una conexión a una base de datos). Están conectados a los dispositivos principales, de tal manera que estos puedan utilizarlos para obtener sus servicios.

Observadores del contexto remoto Se trata de un tipo especial de *observador de contexto* que se encarga de transmitir información entre distintos nodos FAERIE.

La organización explicada anteriormente se corresponde con un modelo de pizarras federadas. Esto quiere decir que, en lugar de existir una única pizarra almacenando el modelo del contexto, hay un conjunto de pizarras que se comunican entre sí para compartir información y crear una única pizarra virtual. De este modo, el modelo no está situado en un único punto susceptible de fallos, y además se permiten mecanismos de replicación para evitar pérdidas de información. En la arquitectura FAERIE, estos elementos

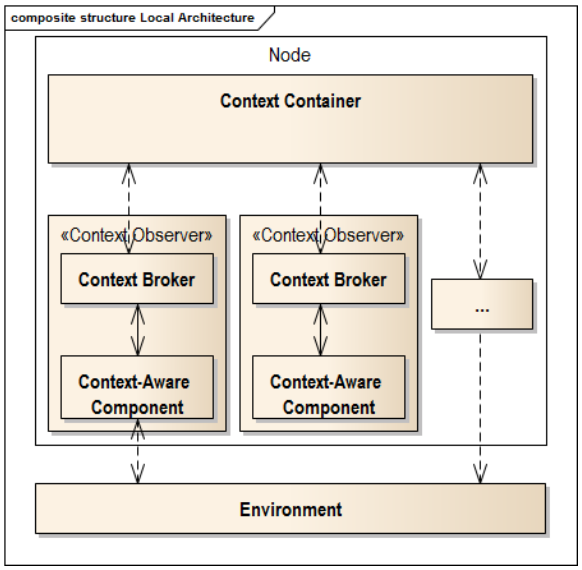

Figura 4.4: Elementos dentro de un nodo FAERIE.

se corresponden con los representados en la Figura 4.4. Dentro de un nodo FAERIE se encuentra el *contenedor de contexto*, que es análogo a la pizarra, y un conjunto de *observadores de contexto*, que son los componentes que la observan y manipulan. De este modo, los *observadores de contexto* son responsables de hacer progresar el estado del contexto a lo largo del tiempo. Están formados a su vez por un *bróker* y por un CSC. Esta organización trata de cumplir el requisito f.

Contenedor del contexto Este elemento gestiona las peticiones de cambios en la representación del contexto, activando los observadores necesarios para establecer los flujos de información. La información del contexto se representa como un modelo de objetos básico, como se explicará en el Capítulo 6, cumpliendo el requisito i.

Componentes sensibles al contexto (CSC) Realizan peticiones al contenedor, tanto para leer información como para modificarla. Además, pueden suscribirse a cambios concretos del contexto para llevar a cabo acciones específicas en esos casos.

Brókers del contexto Son subcomponentes que sirven de puente entre los componentes sensibles al contexto y la plataforma FAERIE. Ofrecen los servicios de acceso al contenedor y también a la capa de infraestructura.

De una manera formal, un *observador del contexto general* actúa como una función de la forma $o_j : K \times P \rightarrow K \times P$. Esto quiere decir que,

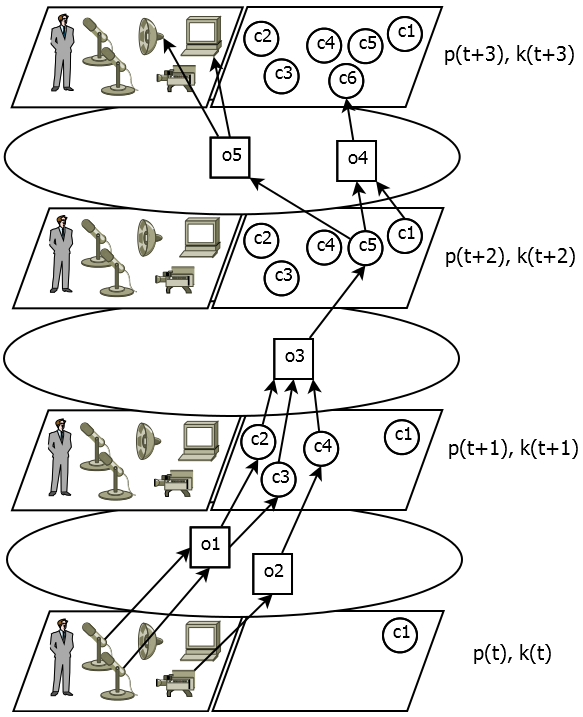

Figura 4.5: Proceso de cambio del contexto a lo largo del tiempo.

dado un estado del entorno físico y el contenedor del contexto, el observador genera un nuevo estado del contenedor del contexto y cambia el entorno físico. Asumiendo una división discreta del tiempo, la Figura 4.5 representa este proceso: $p(t)$ y $k(t)$ son el entorno físico y el estado del contexto en tiempo t; $o1, \ldots, o5$ son observadores del contexto y $c1, \ldots, c6$ son elementos del contexto. El estado del contenedor del contexto en un momento dado se obtiene como una combinación de los resultados de los procesos de los observadores en el instante de tiempo anterior.

Mediante esta organización se cumple el requisito h., ya que las capas de abstracción de información no están delimitadas de ninguna manera preestablecida. Toda la información se almacena por igual en el *contenedor del contexto* y es leída por cualquier *observador del contexto*. Estos componentes se describirán en mayor detalle en el Capítulo 6.

4.4. Conclusiones

Este capítulo describe la visión general de la arquitectura FAERIE, que será refinada en los sucesivos capítulos. La primera perspectiva presenta su organización funcional, es decir, cómo está estructurada la funcionalidad que

ofrece la arquitectura. Ésta se basa en un determinado entorno de ejecución, que puede pertenecer a distintas plataformas. Sobre estas plataformas, se construye la capa de infraestructura de componentes, que tiene como objetivo aislar las aplicaciones de la plataforma concreta donde se ejecutan, y proporcionar algunos servicios de administración. A partir de aquí se construye una serie de componentes que permiten automatizar la coordinación en la gestión del contexto. Por último, se especifican una serie de patrones que permiten definir comportamientos adaptativos, es decir, que cambian según las condiciones del contexto.

Las aplicaciones que se construyen sobre esta infraestructura pueden hacer uso de los servicios de cualquiera de las capas, aunque la capa superior ya ofrece los que la mayoría de las aplicaciones van a necesitar. De este modo, una aplicación definirá un conjunto de comportamientos, posiblemente organizados como actividades en flujos de trabajo, que van cambiando con respecto a ciertos cambios del contexto. Estos comportamientos pueden a su vez declarar sus propios cambios. Además, también se permite definir comportamientos adaptativos genéricos, que tienen que ver con la forma de resolver la redundancia en las fuentes de información.

Después de esto, se describe la visión de la organización de un sistema FAERIE. Éste está compuesto de varios *entornos* organizados alrededor de un *dispositivo principal* que ejecuta la plataforma. Después se distingue entre *entornos físicos*, donde se sitúan los sensores y dispositivos, y *entornos computacionales*, donde se ejecutan los componentes de la plataforma. Estos *entornos computacionales*, también llamados *nodos FAERIE*, están conectados lógicamente unos con otros de tal manera que comparten la información de forma transparente. De este modo, cada componente que utiliza información del contexto no necesita conocer quién la proporciona.

Capítulo 5

Infraestructura de componentes

Las ciencias tienen las raíces amargas,
pero muy dulces los frutos.

Aristóteles.

Resumen: En este capítulo se explica la capa que proporciona los servicios básicos de control del ciclo de vida y la conexión entre componentes FAERIE. En primer lugar se introduce la estructura de la capa, después se explica cómo ésta gestiona los componentes a nivel local y después se describe cómo funciona en un entorno distribuido.

5.1. Introducción

Los componentes software en FAERIE se ejecutan en un entorno computacional distribuido, que controla su ciclo de vida (activación y desactivación), y ofrece servicios de interconexión y arranque de otros componentes. Estos servicios pueden implementarse mediante distintas plataformas, como OSGi o RMI, de forma transparente para los componentes. La presente capa se construye sobre estas plataformas para dotar a los sistemas FAERIE de independencia de las mismas.

Las próximas secciones describen cómo se estructura esta capa. En primer lugar, en la Sección 5.2 se describe cómo es la organización de los elementos en un único entorno. Posteriormente, en la Sección 5.3 se describe cómo se comunican varios de estos entornos.

5.2. Estructura de despliegue local

Una aplicación FAERIE consiste en un conjunto de componentes que colaboran entre sí. Un componente es un artefacto que potencialmente contiene dos partes: una interfaz, que especifica las operaciones que puede llevar a cabo el componente; y una implementación, que consiste en un código que lleva a cabo las operaciones especificadas en la interfaz.

En FAERIE, estos componentes se ejecutan en un *nodo* sobre una cierta plataforma, y un SSC consiste en varios interconectados. Las siguientes subsecciones describen la estructura de un componente, la de un nodo, y cuál es el ciclo de vida de un componente dentro de un nodo.

5.2.1. Estructura de un componente FAERIE

Un componente FAERIE puede ser de dos tipos: de modelo, y de comportamiento. Los componentes de modelo contienen la estructura de la información de un dominio, y los de comportamiento contienen un proceso o conjunto de procesos que se ejecutan en el sistema, y que pueden formar parte de un flujo de trabajo mayor. Todos los componentes tienen una parte específica y otra genérica que se implementa para proporcionar acceso a y desde la plataforma. Su estructura se describe a continuación.

5.2.1.1. Estructura general

Todo componente tiene un ID (IDentificador) con una estructura basada en Maven (Massol y O'Brien, 2005). Utilizando ejemplos de los casos de estudio del Capítulo 11, `es.ucm.fdi.grasia.faerie:userWifiLocator:1.2.7` es un componente que calcula la localización de un usuario a partir de las redes Wi-Fi a las que tiene acceso desde su teléfono móvil. Consiste en los siguientes elementos:

- ID de grupo: representa el espacio de nombres del componente (`es.ucm.fdi.grasia.faerie`).
- ID de artefacto: es el identificador único dentro del espacio de nombres (`userWifiLocator`).
- Versión: la versión del componente. Dos versiones del mismo componente se consideran componentes distintos a todos los efectos (`1.2.7`).

5.2.1.2. Componente de modelo

Un componente de modelo, además de proporcionar el ID explicado anteriormente, especifica otro conjunto de componentes de modelo a los que

extiende y aporta un nuevo conjunto de clases. Estas clases representan únicamente la estructura de la información en un determinado dominio, y no deben tener comportamiento activo. Son utilizadas por componentes de comportamiento para almacenar dicha información y pueden ser extendidas por otros componentes de modelo. Un ejemplo de componente de modelo es `es.ucm.fdi.grasia.faerie:teacherFindingModel:1.2.7`, que agrega un conjunto de entidades involucradas en la aplicación de búsqueda de profesores, como el horario de clases o el estado de la actividad "realizar tutoría". Se describe con mayor detalle en el Capítulo 11.

5.2.1.3. Componente de comportamiento

Un componente de comportamiento, además del ID, contiene los siguientes elementos: un conjunto de servicios proporcionados, cada uno representado por un conjunto de métodos implementados; un conjunto de servicios requeridos; y la implementación que hace uso de los segundos y proporciona los primeros.

Para facilitar la implementación de estos componentes, FAERIE proporciona una clase abstracta (la clase `AbstractContextComponent`), que implementa algunas tareas básicas de la infraestructura. Estas tareas se explican en la Subsección 5.2.3. Un ejemplo de este tipo de componente, es `es.ucm.fdi.grasia.faerie:timetableLocator:1.2.7`, que lleva a cabo el cálculo de la localización de un profesor a partir de su horario de clases.

5.2.1.4. Envoltorio especifico de la plataforma

Cada componente FAERIE de los tipos anteriores, posee un conjunto de elementos específicos de la plataforma en la que se ejecuta (por ejemplo, los ficheros de configuración necesarios para su lanzamiento, o clases que implementan ciertas interfaces obligatorias). Estos elementos son creados en el proceso de construcción, explicado en el capítulo anterior. Además, se proporciona un acceso abstracto a ciertos servicios proporcionados por las plataformas de componentes consideradas, como el lanzamiento de nuevos componentes o la conexión con otros. El uso de estos servicios es necesario, por ejemplo, cuando se quiere monitorizar el sistema o implementar interfaces de administración para arrancar y parar componentes. El subcomponente bróker del contexto da acceso a los mismos, como se explica en la sección 5.2.3.2. Existe una implementación de este bróker para cada una de las plataformas.

5.2.2. Estructura de un nodo FAERIE

Ejemplos de nodos FAERIE son el entorno que ejecuta un dispositivo móvil de un determinado usuario, o el que ejecuta un ordenador en una loca-

lización estática. Ambos podrían conectarse entre sí mediante el acceso a una misma red inalámbrica, formando un SSC mayor. Un nodo está compuesto por los siguientes elementos:

1. Un elemento ejecutable (como un script) que acepta ciertos parámetros iniciales: máxima memoria disponible, puerto de comunicación, etc.
2. Un conjunto de ficheros de configuración de la plataforma subyacente, de FAERIE y de las aplicaciones (p.ej., parámetros de acceso a una base de datos).
3. Los componentes a ejecutar, de la plataforma, de FAERIE y de las aplicaciones (p.ej., el componente de localización por Wi-Fi).

5.2.3. Ciclo de vida de un componente FAERIE

A continuación se describen las tareas que se llevan a cabo durante el arranque de un componente FAERIE y a qué operaciones tiene acceso. Toda la funcionalidad es proporcionada por el bróker del contexto, un componente que se crea para cada componente FAERIE. Sirve de puente de acceso a la plataforma y además se encarga de controlar su ciclo de vida.

5.2.3.1. Activación y desactivación del componente

La primera tarea que hace cada bróker del contexto para el componente que gestiona es intentar resolver sus dependencias haciendo uso de los servicios específicos de la plataforma en uso, es decir, comprobar qué servicios necesita e intentar encontrarlos. Si lo consigue, obtendrá un punto de acceso al servicio, y acto seguido lo inyectará al componente gestionado (i.e., inicializará una variable a la que el componente gestionado tenga acceso). Esto se representa mediante la acción de transición *bindContext*, como muestra la Figura 5.1. Esta fase preliminar se lleva a cabo para considerar posibles cambios en servicios de la plataforma. De este modo, si alguno de los servicios no está disponible en un determinado momento, todos los componentes que dependan de él, deberán pasar a estado *uninitialized*.

Después de esto, el *bróker del contexto* usa los métodos de la interfaz de notificación del `ContextComponent` (ver Listado A.3) para permitir al CSC implementar las tareas que quiere llevar a cabo en los distintos puntos de su ciclo de vida. El método *activate* se llama para disparar el código de inicialización. El método *deactivate* se llama antes del cierre del sistema, ya sea del CSC o del contenedor del contexto, para disparar el código de cierre.

5.2.3.2. Servicios de la plataforma

El bróker del contexto, además de llamar a los métodos de ciclo de vida del componente que gestiona, da acceso a los métodos que se muestran en el

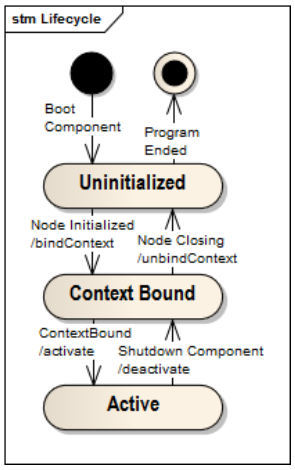

Figura 5.1: Diagrama de estados de un componente en su activación y desactivación.

```
public interface ContextComponent {

  public void activate();

  public void deactivate();
}
```

Listado 5.1: La interfaz *ContextComponent* contiene los métodos de notificación del ciclo de vida del componente.

Listado 5.2.

Los métodos se describen como sigue:

1. **getContext** da acceso a la referencia del contenedor del contexto, lo que permite modificar el modelo del contexto y suscribirse a los cambios. Estos servicios se explican en el Capítulo 6.

2. **installBundle** permite instalar un nuevo componente FAERIE en la plataforma a partir de una URL (*Uniform Resource Locator*) que apunta al artefacto. Se devuelve una referencia que permite desinstalarlo posteriormente.

3. **uninstallBundle** desinstala un componente FAERIE de la plataforma a partir de la referencia obtenida con el método anterior.

4. **shutdownPlatform** apaga la plataforma en el nodo actual, desactivando todos los componentes del mismo.

```
public interface ContextBroker {

  public Context getContext();

  public BundleEntity installBundle(URL bundleURL) IOException;

  public void uninstallBundle(BundleEntity bundleEntity) throws
      IOEXception;

  public void shutdownPlatform() throws IOEXception;
}
```

Listado 5.2: La interfaz `ContextBroker` contiene los métodos de acceso a la plataforma.

5.3. Estructura de despliegue distribuida

Un SSC está compuesto por varios nodos como los descritos en la sección anterior. Cada nodo se identifica por una URI (*Uniform Resource Identifier*), y puede estar ejecutando una plataforma distinta a los demás.

La manera de distribuir los servicios en red de cada plataforma puede ser diferente, pero al final deben obtenerse dos cosas de estos servicios: una referencia de acceso, o *proxy*; y un conjunto de propiedades del servicio. Estas propiedades se ofrecen como una lista de pares clave-valor. En el caso de OSGi, se hace uso del protocolo Zeroconf (Steinberg y Cheshire, 2005), que funciona en redes locales sin necesidad de configuración previa, haciendo uso del protocolo DNS (*Domain Name System*). RMI y Web Services por el contrario necesitan unos componentes extra (aparte de los nodos FAERIE), para poder ofrecer servicios distribuidos. En el caso de RMI (Plasil y Stal, 1998) necesita un registro remoto de interfaces, y en el caso de Web Services (Alonso et al., 2004), el registro UDDI (*Universal Description, Discovery and Integration*).

En FAERIE, sólo va a existir un tipo de componente que va a ofrecer sus servicios de forma remota. Este componente es el *observador del contexto remoto*, el cual se explicará en el siguiente capítulo, y cuyo objetivo es distribuir el modelo del contexto.

Puede ocurrir en un SSC que coexistan distintos nodos FAERIE que ejecuten distintas plataformas. La coexistencia entre estos nodos heterogéneos se gestiona mediante distintos tipos de *observadores del contexto remoto*. Éstos sirven de adaptadores entre distintas plataformas, p.ej., OSGi/RMI u OSGi/Web Services, dependiendo del nodo local y del nodo remoto. Al final, todas las referencias remotas se manejan asociadas al ID del nodo remoto. Como el id de los nodos sí se garantiza que es único en FAERIE, la duplicidad de referencias de acceso queda eliminada.

5.4. Conclusiones

Este capítulo ha descrito la capa de la infraestructura FAERIE que se construye directamente sobre el entorno que ejecuta los componentes del SSC. Su cometido principal es abstraer a las capas superiores de la plataforma concreta que se está utilizando, y proporcionar servicios de administración del sistema. De este modo, los sistemas FAERIE pueden trabajar en distintas plataformas, siempre que la capa de infraestructura sea la adecuada a la misma.

La unidad básica de despliegue de un SSC es el *nodo FAERIE*, que representa un entorno de ejecución donde persiste el sistema. El nodo está poblado por distintos componentes, de modelo o de comportamiento, con una cierta estructura. Estos componentes se desarrollan de forma independiente del entorno de ejecución, y luego son recubiertos por los elementos necesarios para ejecutarlos en ese entorno. Este recubrimiento se puede hacer automáticamente, mediante el proceso de construcción descrito en la Sección 9.2.

En este nivel se define el subcomponente *bróker del contexto*, que está incluido en cada componente FAERIE. Se encarga de activar y desactivar el componente, y de proporcionarle acceso a los servicios de la plataforma. Además, también proporciona acceso a los servicios publicados en la misma, incluyendo aquellos que se encuentran en nodos remotos. Estos nodos remotos pueden ejecutarse en plataformas distintas. Esta circunstancia se maneja mediante componentes adaptadores, que traducen los protocolos utilizados en las diferentes plataformas. La existencia de este componente simplifica la estructura de los componentes de comportamiento.

Capítulo 6

Gestión del contexto

Sólo es útil el conocimiento que nos hace mejores.

Frase atribuida a Sócrates.

Resumen: Este capítulo explica la capa de funcionalidad de gestión del contexto de la arquitectura FAERIE. En primer lugar se muestra una visión general de las tareas de gestión del contexto. Luego se explica el modelado de la información y finalmente las cuestiones de uso y procesamiento.

6.1. Introducción

FAERIE utiliza un enfoque orientado a objetos (Strang y Linnhoff-Popien, 2004) basado en el patrón *observador* para procesar los cambios que se producen en el contexto y en el entorno. La idea es que los distintos elementos del modelo del contexto tienen un conjunto de *observadores*, que responden a los cambios que estos sufren para llevar a cabo las tareas correspondientes. Por ejemplo, un componente que procesa la localización de un usuario a partir de la información de unos sensores, es un observador que se activa cada vez que hay un cambio significativo en la información proporcionada por los mismos. La figura 6.1 muestra los elementos que pueden encontrarse en el espacio computacional de un entorno FAERIE. Aquellos relativos al modelado de información se encuentran en la mitad derecha, mientras que los relativos a su procesamiento se sitúan a la izquierda. Implementaciones concretas de estos elementos pueden encontrarse en el Capítulo 11.

El *contenedor del contexto* (`ContextContainer`), como se introdujo en el Capítulo 4, es un componente que almacena la información contextual en

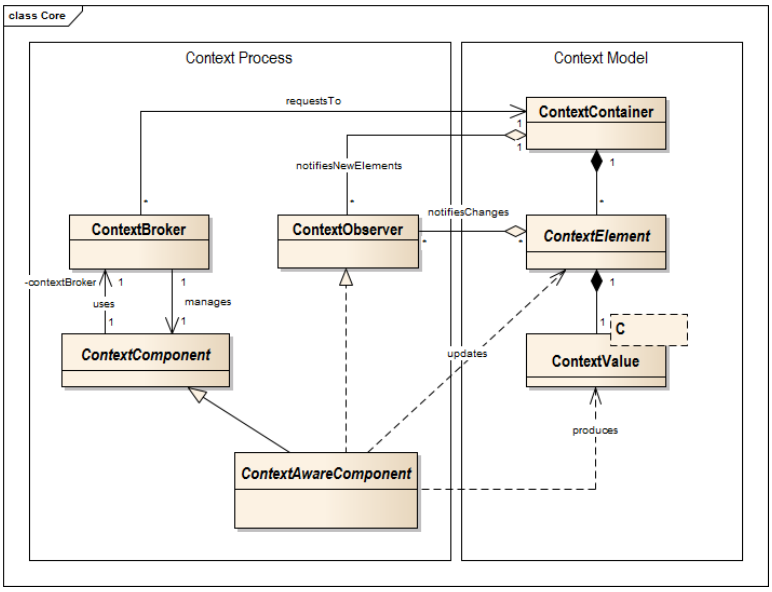

Figura 6.1: Elementos básicos para la gestión del contexto.

forma de *elementos del contexto* (`ContextElement`), que poseen un determinado *valor* en un momento del tiempo (`ContextValue`). Los elementos del contexto se comportan como elementos *observables*, de tal manera que otros componentes pueden suscribirse a sus cambios para ser notificados cuando estos ocurran. La interfaz que acepta estos eventos es la de *observador del contexto* (`ContextObserver`). Los componentes que implementan esta interfaz dan acceso a métodos de notificación cuando aparecen nuevos elementos en el contexto, o cuando alguno de ellos cambia su estado. El *bróker del contexto* (`ContextBroker`) se encarga de dar acceso a los servicios de la plataforma y a los del *contenedor del contexto*. Además, notifica al *componente del contexto* (`ContextComponent`) acerca de los eventos en su ciclo de vida. Finalmente, el CSC (`ContextAwareComponent`) es aquel que es implementado para aplicar las reglas de negocio de un SSC concreto (ver Sección 4.3).

6.2. Modelado y representación del contexto

En esta sección se cuenta cómo se manipulan los *elementos del contexto*. En primer lugar se describe su estructura básica, y después un modelo base que sirve de punto de partida para crear nuevos modelos. Un elemento del contexto es un fragmento del modelo que representa el contexto de la aplicación. Un ejemplo de elemento es una entidad *Usuario*, que puede estar unida a otros elementos, como la relación *Localización* o el atributo *Nombre*. La manipulación de estos elementos consiste en observar cómo cambian sus

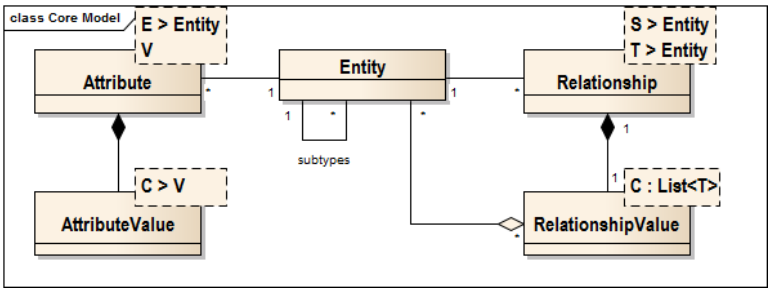

Figura 6.2: Superclases de modelado del contexto.

elementos relacionados y los valores de sus atributos, o modificarlos con respecto a una lógica. Por ejemplo, observar la localización de un determinado usuario para determinar su atributo *EstaEnCasa*.

6.2.1. Estructura fundamental

Cada *elemento del contexto* en un momento dado se representa como un `ContextElement` de alguno de los tres tipos posibles: *entidad* (`Entity`), *atributo* (`Attribute`) y *relación* (`Relationship`), como muestra la Figura 6.2. Los tipos V, E, C, S y T representan clases cualesquiera, aunque en el caso de E, S y T, descienden del tipo `Entity`, mientras que C en `Attribute` desciende de V, y C en `Relationship` es una lista de elementos del tipo T. Una *entidad* representa un concepto del vocabulario del dominio del sistema (e.g., una persona, lugar u objeto), y agrupa todos los *atributos* concretos y las *relaciones* asociadas con él. Un *atributo* aplicado mantiene alguna información de tipo V con un cierto significado relacionada con una *entidad* de tipo E (e.g., la edad en años de una persona, o la cadena nombre de un lugar). Una *relación* enlaza una *entidad* de tipo S con otras de tipo T de acuerdo a cierta semántica (e.g., una persona con el lugar donde está localizada). El estado actual de cada *atributo* y *relación* se almacena como un *valor de contexto* (`ContextValue`). Un *valor de contexto* contiene, aparte de un dato de un cierto tipo C (especificado por un *atributo* o *relación* concreto), una referencia al *observador del contexto* (`ContextObserver`) fuente (i.e., el observador del contexto que produjo el valor), su tiempo de creación (i.e., el momento en el que el valor fue obtenido), y metadatos adicionales (e.g., un número representando la calidad o coste de la medida).

De una manera formal, los contenidos de un *contenedor de contexto* κ en un tiempo t, $\kappa(t) \in K$ (definiciones en el capítulo 4), $\kappa(t) \subset ContextElement$ son de la forma $\kappa(t) = E(t) \cup \mathcal{A}(t) \cup \mathcal{R}(t) \cup O(t)$, donde:

- $E(t) \subset Entity$ es el conjunto finito de entidades referenciadas en tiempo t, $e_1, \ldots, e_r$.

- Sea $A : Entity \times Attribute$ el conjunto de pares de entidades y atributos. $\mathcal{A}(t) : A \rightarrow ContextValue$ es una función tal que $\mathcal{A}(t)(e, \alpha) = v$ siendo v el valor del atributo α de la entidad e en tiempo t.

- Sea $R : Entity \times Relationship$ el conjunto de pares de entidades y relaciones. $\mathcal{R}(t) : R \rightarrow [ContextValue]$ es una función tal que $\mathcal{R}(t)(e, \rho) = [v]$ siendo $[v]$ la lista de entidades relacionadas con e mediante la relación ρ en tiempo t.

- $O(t) \subset ContextObserver$ es el conjunto finito de observadores de contexto existentes en tiempo t, $o_1, \ldots, o_p$. Cada observador o_j tiene un conjunto de elementos de contexto que está *observando* en tiempo t, $Read_j(t) \subseteq \kappa(t)$, y un conjunto de elementos de contexto que está *actualizando* en tiempo t, $Write_j(t) \subseteq \kappa(t)$

Los *ContextElement* proporcionan un método para hacer un *intento de actualización* de sus valores. El listado A.4 muestra su pseudocódigo. Si el intento tiene éxito o no depende de los mecanismos de consistencia implementados por el *contenedor*, como se explicará en la Sección 6.3.4. De esta manera, varios observadores del contexto pueden tener los mismos *Attributes* y *Relationships* en su conjunto $Write_j(t)$ en un cierto tiempo, pero solo algunos de ellos realmente actualizarán su valor. Tanto los intentos como las actualizaciones reales generarán eventos a los observadores del contexto suscritos. Usando esta información, cada observador de contexto es capaz de conocer qué valores están generando los otros, y usarlos en su propio procesamiento.

```
public synchronized boolean attemptUpdate(
    ContextValue<T> newValue) {
  boolean succeeded = false;
  // Notifica el intento al contexto
  if (this.getContext().publishValue(this, newValue)) {
    // Si este intento se acepta, el valor se actualiza
    this.value = newValue;
    succeeded = true;
  }
  // Notifica a los observadores, haya éxito o no
  this.notifyAllObservers(UPDATE_ATTEMPT, newValue);
  return succeeded;
}
```

Listado 6.1: Método de actualización de un *Attribute*.

El Listado 6.2 muestra otros dos métodos que ofrecen los `ContextElement`. Estos dos métodos sirven para gestionar los `ContextObserver`s que son notificados ante los cambios de un determinado elemento del contexto, tal y como se hace en el listado anterior.

```
public void subscribe(ContextObserver observer);

public void unsubscribe(ContextObserver observer);
```

Listado 6.2: Método de subscripción *ContextElement*.

6.2.2. Modelo base

La estructura fundamental del contexto puede extenderse a través de herencia, especificando nuevos tipos de entidades, relaciones y atributos. Estos nuevos tipos pueden contener restricciones específicas en la información que contienen e incluso especificar métodos con sus propias semánticas.

FAERIE proporciona un modelo base que contiene, sobre la estructura fundamental, los siguientes tipos de entidades y relaciones:

- Relación `has`: esta relación se utiliza para indicar que una cierta entidad contiene a otras entidades.
- Entidad `Environment`: esta entidad es una entidad única en el nodo, identificada mediante un ID global, que contiene todas las entidades referenciadas en el nodo, mediante la relación `has`.
- Entidad `ConditionSet`: esta entidad se inicializa con un conjunto de condiciones. El contenido de la entidad es igual a todas las entidades existentes en el nodo que cumplen las condiciones especificadas. Las condiciones son objetos que proporcionan un método `resolve` que, dado un `ContextElement` devuelve `true` si cumplen la condición especificada o `false` en otro caso. Ejemplos de condiciones son los siguientes:
  - `ClassCondition`: el elemento es de una clase determinada.
  - `AttributeCondition`: el elemento tiene un atributo determinado que cumple a su vez una condición.
  - `RelationshipCondition`: el elemento tiene una relación determinada que cumple a su vez una condición.

6.3. Procesamiento sensible al contexto

En esta sección se cuenta cómo se lleva a cabo el procesamiento sensible al contexto, es decir, cómo se obtiene el contexto a partir de distintas fuentes, y qué se hace con él. En primer lugar, en la Sección 6.3.1, se describe qué métodos ofrece el *bróker del contexto* para estas tareas, y los que deben implementarse en cada observador del contexto. Después, en la Sección 6.3.2, se explica cómo se transmiten los cambios en el contexto a los distintos elementos interesados. Finalmente se explican las tres tareas básicas que se

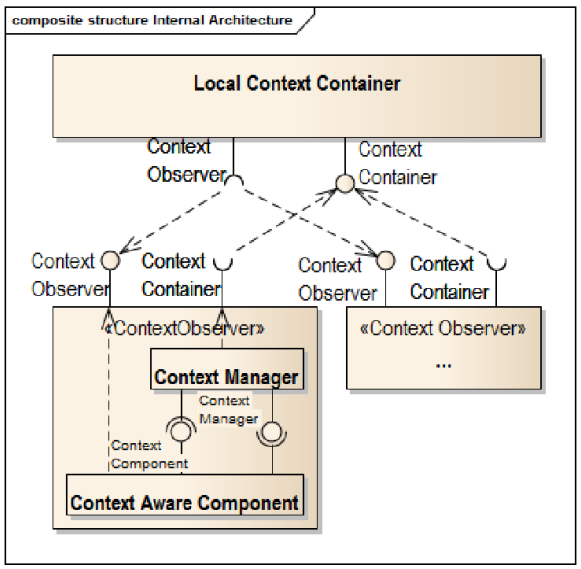

Figura 6.3: Dependencias entre las interfaces de los componentes FAERIE.

hacen con el contexto: percepción, actuación y procesamiento. La percepción, en la Sección 6.3.3.1, explica de qué forma se percibe y actualiza el contexto. La actuación, en la Sección 6.3.3.2, explica de qué forma se actúa sobre el entorno a partir de cambios en el contexto. El procesamiento, en la Sección 6.3.4, explica cómo se transforma el contexto.

6.3.1. Interfaz de uso

Los componentes software que se ejecutan en un nodo FAERIE deben implementar ciertas interfaces para poder colaborar. La Figura 6.3 muestra las dependencias entre ellas.

El *contenedor del contexto* proporciona métodos de acceso a la representación orientada a objetos del contexto actual (ver Sección 6.2). También asocia cada elemento de esa representación, con los *observadores del contexto* que trabajan en él (ver Sección 6.3.2). El *contenedor del contexto* ofrece estos servicios a través de la interfaz *ContextContainer*, y requiere que los componentes interesados ofrezcan la interfaz *ContextObserver* para informarles de los cambios en el contexto.

Los componentes *observadores* solicitan y reaccionan a cambios en la representación del contexto actual. Pueden clasificarse como *observadores anclados* u *observadores de abstracción*. Los *observadores anclados* acceden al entorno exterior usando *dispositivos periféricos*, ya sea para modificarlo de acuerdo al contexto, o para observarlo y actualizar el contexto de acuerdo a él. Más concretamente, los sensores pueden ser vistos como funciones de la

forma $o_{sensor} : K \times P \rightarrow K$, y los actuadores como $o_{actuator} : K \rightarrow K \times P$. Los *observadores de abstracción* no acceden al entorno, ya que solo actualizan la representación del contexto siguiendo su lógica interna y usando otra información ya presente en el contexto. En otras palabras, los *observadores de abstracción* son de la forma $o_{abs} : K \rightarrow K$. Esta actualización puede consistir en un proceso de fusión de información (ver Sección 6.3.4). Ambos tipos ofrecen la interfaz *ContextObserver* y utilizan la interfaz *ContextContainer*, que los enlaza a los contenedores de contexto. Internamente están divididos en dos subcomponentes: el *bróker del contexto* y el CSC.

El subcomponente *bróker del contexto* implementa la interfaz *ContextBroker* para cumplir varias responsabilidades, tal como se explica en el Capítulo 5. La primera, adaptar la interfaz *ContextContainer* para ofrecer al componente la misma funcionalidad pero de forma más simple. Esto libera al CSC de algunos asuntos de gestión, tales como identificarse en cada petición de información. De esta manera puede trabajar como si fuese el único componente que está accediendo al contexto. La segunda, obtener y mantener controlada la referencia al *contenedor del contexto* de su *nodo*. Dado que el *contenedor de contexto* es un componente dinámico, puede ser desactivado o cambiado en tiempo de ejecución. El *bróker del contexto* oculta esto al CSC. La tercera, contener el código necesario para arrancar el *observador de contexto* dentro del *nodo*. Este proceso proporciona la referencia al componente *observador de contexto* para el *contenedor*.

```
public <T extends ContextElement> T request(Class<T>
    elementClass, URI elementId);

public boolean release(URI elementId);
```

Listado 6.3: Métodos de acceso al contexto del *ContextBroker*.

El Listado 6.3 muestra los métodos de la interfaz *ContextBroker* que permiten al CSC manipular la información de contexto. El método *request* recupera o publica un cierto *elemento del contexto* en el *contenedor*. Dada una clase de *ContextElement* como parámetro del método, si el *contenedor de contexto* contiene una instancia de esta clase con el mismo identificador, entonces la devuelve; de lo contrario, se crea una nueva instancia que se publica en el *contenedor* y la devuelve. Esta acción crea un *enlace activo* del *elemento del contexto* con el solicitante. Al publicar se notifica a cada *observador* capaz de *manejar* la instancia sobre esta agregación, para permitirle realizar un *enlace pasivo* con el nuevo *ContextElement* como consumidor o proveedor.

Así, la diferencia entre un *enlace activo* y en *enlace pasivo* es la siguiente: un *enlace activo* se crea cuando un componente hace *request* de un elemento del contexto, e indica que el componente necesita ese elemento para lo que

está haciendo; por el contrario, un *enlace pasivo* se crea cuando un componente es notificado por el contenedor de la petición de un elemento del contexto, e indica que el componente está reaccionando para utilizar su valor o para calcularlo. De este modo, el elemento sigue en el contexto mientras haya *enlaces activos* al mismo, y cuando deja de haberlos (todos los componentes ejecutan *release*), se deshacen todos los *enlaces pasivos*, liberando los recursos que ya no son necesarios.

Los componentes usan el método *release* sobre un elemento de información con el que tienen un *enlace activo* para declarar que no lo necesitan más. Si no hay otro componente con un *enlace activo* en el mismo elemento del contexto, el contenedor lo retira. Retirar significa notificar a cada *observador* con un *enlace pasivo* con el *ContextElement* sobre esta retirada. De este modo, los observadores pueden liberar los recursos que ya no se necesitan, por ejemplo, dejar de monitorizar un sensor.

El CSC implementa dos interfaces: *ContextObserver* para reaccionar a los cambios en el contexto, y la ya mencionada *ContextComponent* (ver Sección 5.2.3.1) para seguir su propio ciclo de vida.

```
public int handles(ContextEvent event);

public void elementAdded(ContextEvent event);

public void elementRemoved(ContextEvent event);

public void elementUpdated(ContextEvent event);
```

Listado 6.4: Métodos de notificación de la interfaz `ContextObserver`.

El Listado A.2 muestra los métodos de la interfaz *ContextObserver*. El método *handles* se invoca para determinar si el componente es capaz de manejar el evento especificado. Un *ContextEvent* puede referirse a una agregación, borrado o actualización de *ContextElement*. Este método devuelve cero si no maneja el evento, o `HANDLE_R`, `HANDLE_W` o `HANDLE_RW`. `HANDLE_R` significa que el observador no modifica el elemento, pero está interesado en sus cambios; `HANDLE_W` implica que el observador modifica el elemento, pero no lee su estado; `HANDLE_RW` es una combinación de los dos anteriores. Cuando este método devuelve algo distinto de cero, los métodos *elementAdded*, *elementRemoved*, y *elementUpdated* pueden llamarse para notificar al componente de los eventos correspondientes. De este modo, los *componentes sensibles* son capaces de seguir los cambios en el contexto.

6.3.2. Difusión del contexto

La difusión del contexto es el proceso que conecta *observadores de contexto* que trabajan con el mismo *elemento de contexto*. Esta conexión puede

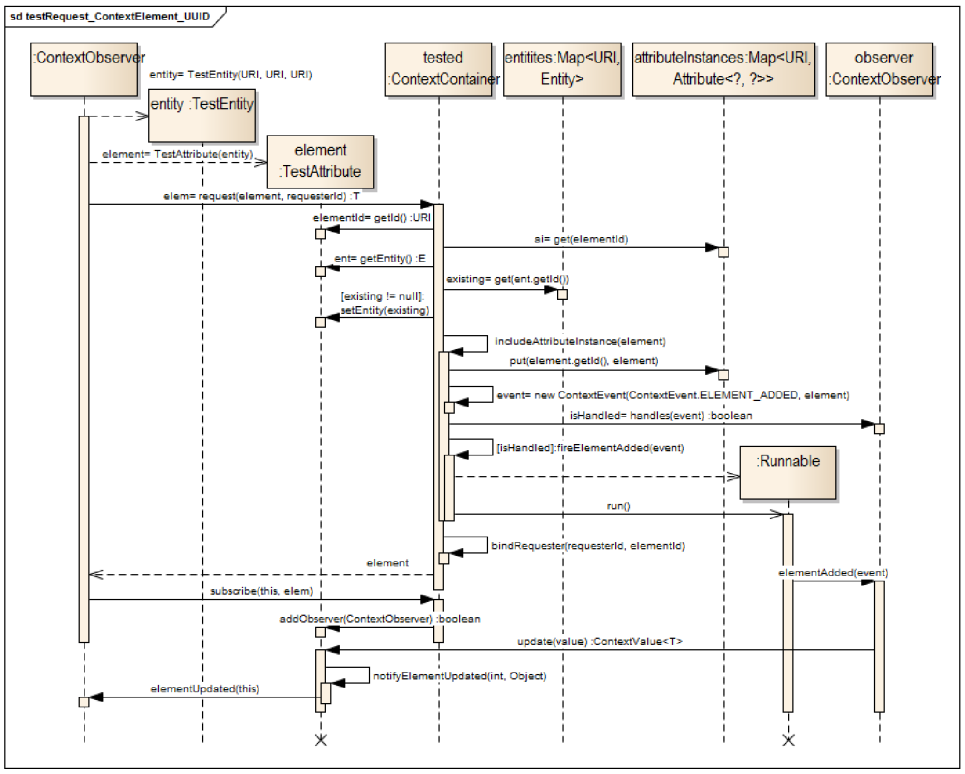

Figura 6.4: Difusión del evento `request` dentro de un nodo.

ocurrir tanto al leer como al actualizar información. En el primer caso, el observador consumidor o_i solicita un $c_a \in Read_i(t)$, de modo que el sistema necesita encontrar otro observador o_j tal que $c_a \in Write_j(t)$ (i.e., proporciona la información requerida). En el segundo caso, el actualizador o_i solicita un $c_a \in Write_i(t)$, de modo que el sistema necesita encontrar otro observador o_j tal que $c_a \in Read_j(t)$ (i.e., que propaga los cambios, probablemente bajando el nivel de abstracción). Hay dos niveles de difusión del contexto: *local*, cuando ocurre dentro de un nodo, y *remoto*, que involucra varios nodos.

La figura 6.4 muestra un ejemplo del proceso de *difusión del contexto local*. Un *observador del contexto* que necesita un elemento del contexto hace una petición al contenedor de contexto mediante su interfaz *ContextContainer*. En este ejemplo, esa petición es para el atributo *TestAttribute* de una cierta *TestEntity* (un ejemplo podría ser el *ValorDetectado* de un cierto *Sensor*). Cuando el contenedor recibe la petición, no tiene elementos previos con las características solicitadas. Por tanto, incluye el proporcionado dentro del contexto e intenta localizar a un *ContextObserver* adecuado para proporcionar su información. El contenedor busca este *ContextObserver* entre aquellos que tiene registrados. Invoca el método *handles* de los observadores usando

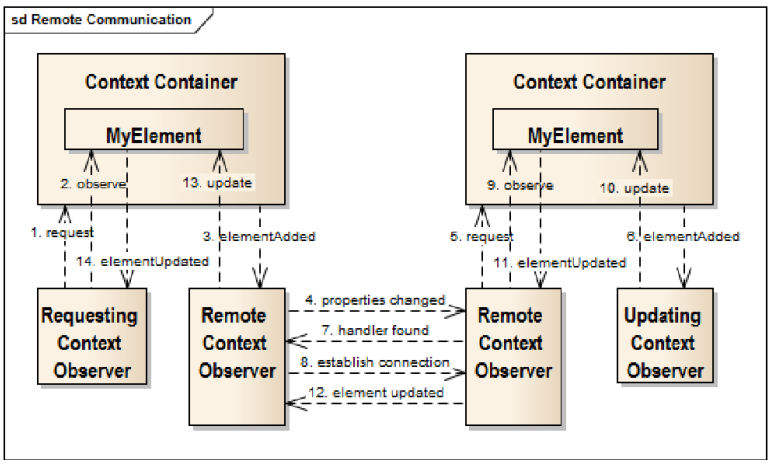

Figura 6.5: Difusión del evento `request` entre varios nodos.

el nuevo elemento como argumento para el evento. El contenedor selecciona el primer observador cuyo método devuelve `HANDLE_W` y lo asocia al elemento utilizando su método *elementAdded*. Desde ese momento, el observador enlazado puede hacer *intento de actualización* sobre el *ContextElement* cada vez que sea necesario. El *ContextElement* notificará directamente estas actualizaciones a todos los *ContextObserver* suscritos a estos eventos. En el caso en el que el contenedor esté configurado para ello, redirigirá la petición al *observador del contexto remoto* del *nodo* si es necesario. Además, los *ContextObserver* que se registren nuevos, serán notificados de la existencia de este elemento.

Un *observador de contexto remoto* lleva a cabo dos tareas específicas en su activación. Difunde en la red una referencia a sí mismo y detecta las referencias y propiedades de otros *observadores de contexto remotos* en la red. Estas propiedades describen los *ContextElement* que estos observadores esperan que otros manejen. Este intercambio de referencias y propiedades soporta la *difusión del contexto remota*, que permite a los observadores remotos actuar como proxies los unos de los otros.

La figura 6.5 muestra el proceso de *difusión remota*. Comienza con una petición que ha sido redirigida a un *observador del contexto remoto*. Este observador modifica sus propiedades registradas para indicar el *ContextElement* que necesita ser manejado. Entonces difunde estos cambios al resto de los *observadores del contexto remoto*. Si alguno de sus *nodos* contiene un componente capaz de manejar dicho *ContextElement*, entonces sus *observadores de contexto remoto* lo notifican de vuelta al solicitante original. Éste escoge uno (o varios) de ellos y establece una conexión entre la copia local y la remota del *ContextElement*. Desde ese momento, cada cambio en la copia local o remota del *ContextElement* será redirigida a través de los

observadores del contexto remoto escogidos.

El uso de *observadores de contexto remoto* hace transparente si actualizan el contexto componentes locales o remotos. El *observador de contexto remoto* actúa como cualquier otro observador en las interacciones locales, mientras que de forma efectiva maneja cada elemento de contexto que debe ser procesado de forma remota mediante colaboración con otros *observadores de contexto remoto*. Esta conexión crea una "superposición virtual" entre contenedores de contexto, permitiendo al sistema trabajar como si estuviese conectado a una única pizarra virtual distribuida.

El protocolo subyacente utilizado para implementar referencias remotas y difusión de propiedades en las implementaciones realizadas en este trabajo es Zeroconf (Steinberg y Cheshire, 2005) (también conocido como *Rendezvous* o *Bonjour*), que utiliza mecanismos DNS. Zeroconf permite publicar y descubrir objetos remotos con propiedades asociadas en la red local. No requiere una configuración especial de los parámetros de la red. Este protocolo es implementado de forma transparente por R-OSGi (*Remote OSGi*) (Rellermeyer et al., 2007) y por Web Services, infraestructuras que han sido utilizadas en las implementaciones de referencia de FAERIE. Debido a las limitaciones del protocolo Zeroconf, los mecanismos descritos en esta sección sólo funcionan cuando los *nodos* están localizados en la misma red local. Cuando éste no es el caso, es necesario considerar un protocolo distinto.

6.3.3. Percepción y actuación sobre el contexto

La percepción y actuación son ambos extremos de la interacción externa del SSC con el entorno. Percepción se refiere al procesamiento de nuevos datos de los sensores, y actuación al envío de nuevas órdenes a los actuadores. Desde el punto de vista de FAERIE, ambos procesos son muy similares. Comienzan como parte de un proceso de difusión, cuando el contenedor encuentra un *observador de contexto anclado* adecuado para un nuevo *ContextElement*. Entonces, estos procesos sólo difieren en qué componente actualiza el *ContextElement* y cuál lo observa.

6.3.3.1. Percepción

Para la *percepción*, el *observador de contexto* maneja un sensor. El contenedor asocia ese observador al *ContextElement* usando su método *elementAdded*. Esta llamada comienza de forma efectiva la monitorización del sensor (ver listado A.5). El observador crea una hebra donde comprueba cambios en el valor del sensor, y actualiza el *ContextElement* relacionado con él cuando estos cambios ocurren. De la misma forma, el método *elementRemoved* se usa para notificar al observador que el *ContextElement* no se necesita más, y por tanto que pare su monitorización.

De una manera formal, el proceso de percepción actúa como una fun-

```
@Override
public void elementAdded(ContextEvent event) {
  final Attribute attrib = event.getAttributeInstanceArg();
  thread = new Thread(new Runnable() {

    @Override
    public void run() {
      while (true) {
        // Obtain attribute value from the environment
        // ...
        if (/* value has changed */)
          attrib.update(new DefaultContextValue(value, 1.0));
      }
    }
  });
  thread.start();
}

@Override
public void elementRemoved(ContextEvent event) {
  thread.interrupt();
}
```

Listado 6.5: Código que implementa un *observador del contexto* manejando un sensor.

ción de la forma $o_{sense}(t) : K \times P \rightarrow K$ tal que $o_{sense}(t)(\{(e, \alpha, \_)\}, p) = \{(e, \alpha, v)\}$ donde v es el valor obtenido por el observador del contexto o para el atributo α de la entidad e bajo el entorno físico p en el tiempo t. La entidad e representa un dispositivo físico d controlado por el observador.

6.3.3.2. Actuación

El proceso de *actuación*, como el de percepción, comienza con un nuevo *ContextElement*, pero aquí, el proceso dispara una orden para un actuador. Cuando un *observador de contexto* proveedor publica alguno de estos *ContextElement*, el contenedor busca un *observador de contexto* adecuado utilizando el método *handles*. Cuando lo encuentra, invoca el método *elementAdded* del mismo (ver listado A.6). Dado que este último componente maneja un actuador, no actualiza el *ContextElement*, sino que lo *observa*. Entonces, cuando el *ContextElement* es actualizado, el actuador observador recibe la llamada *elementUpdated* (ver listado A.6), y lleva a cabo la acción correspondiente al valor obtenido del elemento.

De una manera formal, el proceso de actuación se puede representar como una función de la forma $o_{act}(t) : K \rightarrow K \times P$ tal que $o_{act}(\{(e, \alpha, v)\}) = (\{(e, \alpha, v)\}, p)$, donde p es el comportamiento ejecutado por el observador o en el entorno en tiempo t como respuesta al valor v del atributo α de la

```
@Override
public void elementAdded(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  attrib.subscribe(this);
}

@Override
public void elementUpdated(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  ContextValue value = attrib.getValue();
  // Check conditions on the obtained value
  // Perform the required actions
  // ...
}

@Override
public void elementRemoved(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  attrib.unsubscribe(this);
}
```

Listado 6.6: Código que implementa un *observador del contexto* manejando un actuador.

entidad e.

6.3.4. Procesamiento del contexto

El procesamiento del contexto puede verse como una combinación de percepción y actuación, fluyendo a través de distintos observadores en dos posibles direcciones: *bottom-up* y *top-down*. La *bottom-up* consiste en traer valores del entorno físico y representarlos en el contexto. La *top-down*, por el contrario, consiste en reflejar valores del contexto en el entorno físico. La diferencia entre los procesos de percepción y actuación, y los de procesamiento del contexto, es que estos últimos consideran también *observadores de abstracción*, en lugar de únicamente *observadores anclados*.

6.3.4.1. Evaluación *bottom-up*

Una evaluación *bottom-up* comienza cuando un *observador* solicita uno o varios *ContextElement* para *observar* sus valores, como en un proceso de *percepción*. En este caso, el *observador de contexto* proveedor puede necesitar peticiones subsecuentes al contexto, produciendo una evaluación en cascada. La evaluación en cascada finaliza en *observadores anclados*. A partir de los valores que éstos proporcionan, los cálculos pendientes se van haciendo hacia atrás. Estos cálculos pueden consistir en *agregación* o *derivación* de valores (Baldauf et al., 2007). La Figura 6.6 muestra un ejemplo de las relaciones en

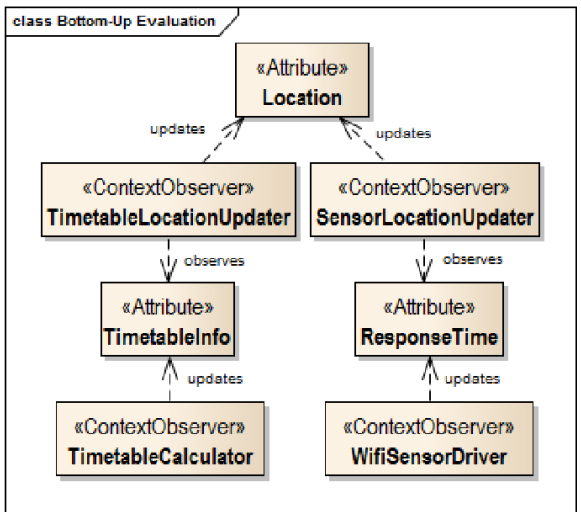

Figura 6.6: Relaciones que se establecen en tiempo de ejecución en una proceso de evaluación *bottom-up*.

ejecución que se establecen durante este proceso.

El Listado A.7 muestra una implementación de un observador envuelto en una evaluación *bottom-up*. Cuando se publica un cierto *Attribute*, se llama al método *elementAdded*. El cálculo de este *Attribute* necesita los valores de dos *Attribute* más, *s1att* y *s2att*, que en este caso provienen de sensores con identificadores *s1* y *s2*. Éstos se solicitan invocando a *request*, y posteriormente se invoca a *subscribe* para observar sus cambios. De este modo, cuando alguno de estos atributos cambia, el método *elementUpdated* es llamado. Aquí se agregan sus valores para calcular el primer *Attribute*. Cuando este *Attribute* de nivel superior se borra del contexto, se llama al método *elementRemoved*. Éste invoca el método *release*, liberando los *Attribute* de nivel inferior utilizados anteriormente para calcular el otro atributo.

De una manera formal, la evaluación *bottom-up* hecha por un observador es una función de la forma $o_{bottomup}(t) : K \rightarrow K$ tal que $o_{bottomup}(t)(\{c_0, \ldots, c_n\}) = \{c_0, \ldots, c_n, c_{n+1}\})$ donde c_{n+1} es el elemento del contexto obtenido por el observador usando una lista de n elementos de contexto $c_0, \ldots, c_n$ presentes en el contenedor.

Un tipo especial de evaluación bottom-up es la fusión de información. Los siguientes son los posibles tipos de fusión de información de contexto:

- **Agregación:** este tipo de procesamiento toma múltiples elementos de contexto y produce un nuevo elemento de contexto que sintetiza toda la información de una forma más condensada o abstracta. Por ejemplo, construir un mapa global usando la información de adyacencia de distintos lugares. Técnicas de agregación típicas aquí pertenecen al campo

```
@Override
public void elementAdded(ContextEvent event) {
  updatedAttribute = event.getAttributeInstanceArg();

  // Request the values of lower context elements
  s1 = contextBroker.request(MySensor.class,
    /* ID of the sensor */);
  s1att = s1.requestAttribute(MySensorAttribute.class);

  s2 = contextBroker.request(MyOtherSensor.class,
    /* ID of the other sensor */);
  s2att = s2.requestAttribute(MyOtherSensorAttribute.class);

  // ...

  // Observe the values of the lower sensors
  s1att.subscribe(this);
  s2att.subscribe(this);
  // ...
}

@Override
public void elementUpdated(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  if (attrib.equals(s1att))
    value1 = attrib.getValue();
  if (attrib.equals(s2att))
    value2 = attrib.getValue();
  // ...

  // Recalculate the context value for the updated attribute,
  // depending on the obtained values
  // ...
  updatedAttribute.update(new DefaultContextValue(value, 1.0));
}

@Override
public void elementRemoved(ContextEvent event) {
  s1att.unsubscribe(this);
  contextBroker.release(s1);
  s2att.unsubscribe(this);
  contextBroker.release(s2);
  // ...
}
```

Listado 6.7: Código para impementar una evaluación *bottom-up*.

del aprendizaje máquina, como el clasificador de Naïve Bayes (resume grandes cantidades de datos como un conjunto de clases definidas por la probabilidad de ciertos rasgos) (West y Harrison, 1997).

- **Derivación:** este tipo de procesamiento es similar a la agregación, pero el resultado es nueva información que es inferida a partir de la información de origen, en lugar de ser una versión sintetizada de la misma. Por ejemplo, deducir la localización de un objeto usando los valores de ciertos sensores, y la información de su despliegue. La inferencia por subsunción es una técnica de interpretación típica (definir un concepto como la conjunción de otros conceptos más simples) (Kokar et al., 2009).
- **Manejo de redundancia/consistencia:** este tipo de procesamiento involucra el manejo de distintas fuentes para la misma información con el objetivo de producir un valor "mejor". El significado de "mejor" depende de ciertas directivas del sistema. Por ejemplo, si hay dos componentes calculando la localización del mismo objeto usando distintos algoritmos y recursos, este proceso puede tomar una de las medidas, o una combinación de ambas. Las técnicas de compresión se usan para tratar con este tipo de problemas (Taubman y Marcellin, 2001).

La información generada en estos procesos es consumida por las aplicaciones sensibles al contexto para llevar a cabo *adaptación* (i.e., modificar el comportamiento externo de forma acorde). Un ejemplo de esto es guiar a los usuarios en un área dependiendo de sus localizaciones actuales.

6.3.4.2. Evaluación *top-down*

La evaluación *top-down* funciona en el sentido opuesto. Un *observador del contexto* solicita *cambiar* uno o más *ContextElement*. Esto causa sucesivas peticiones de *cambio* de otros *ContextElement*. Estas peticiones finalizan en procesos de *actuación*. Entonces, cada vez que el valor del *ContextElement* original de mayor nivel cambia, estos cambios son notificados hacia abajo, activando en última instancia los actuadores. Por este motivo, el proceso se describe como *top-down*. Esto conforma el proceso de propagación de los valores de *ContextElement* hacia valores más concretos. La Figura 6.7 muestra un ejemplo de las relaciones que se forman en tiempo de ejecución durante este proceso.

El Listado A.8 muestra un ejemplo de implementación de un observador de contexto para llevar a cabo una evaluación *top-down*. Cuando se publica un cierto *observedAttribute*, se llama al método *elementAdded*. Éste agrega al *observador de contexto* como observador del parámetro *observedAttribute* de mayor nivel usando el método *subscribe*. Este *Attribute* proporciona un valor que debe propagarse a otros *Attribute* de los que dependen actuadores.

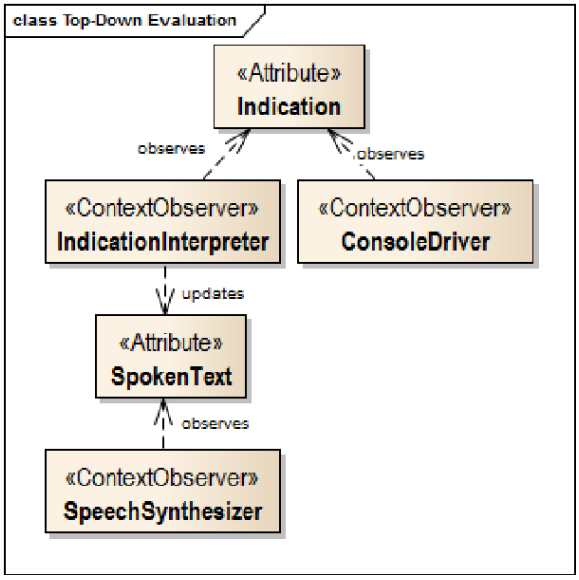

Figura 6.7: Relaciones que se establecen en tiempo de ejecución en una proceso de evaluación *top-down*.

Los *Attribute* *a1att* y *a2att* se solicitan, pero el que se observa es el *Attribute* detonante. Cuando este *Attribute* original cambia, se llama al método *elementUpdated*, y su valor se propaga para actualizar el valor de los otros dos *Attribute*. Cuando el *Attribute* de mayor nivel se borra del contexto, se llama al método *elementRemoved*, el cual libera los *Attribute* de menor nivel que dependen del valor del *Attribute* parámetro.

De una manera formal, la evaluación top-down hecha por un observador es una función de la forma $o_{topdown}(t) : K \rightarrow K$ tal que $o_{topdown}(t)(\{c_0\}) = \{c_0, c_1, \ldots, c_n\}$ donde $c_1, \ldots, c_n$ son los elementos de contexto obtenidos por el observador usando el elemento c_0. Como se puede ver, este proceso "descomprime" c_0 en una lista de elementos de contexto, ya que es el proceso inverso del *bottom-up*.

6.4. Conclusiones

Esta capa de la infraestructura FAERIE gestiona los distintos mecanismos necesarios para modelar, obtener y difundir el contexto en un SSC. El modelo del contexto se construye a partir de tres elementos básicos: *entidades*, *relaciones* y *atributos*. Estos elementos funcionan con el patrón *observador* para notificar los cambios que registran. El modelo básico es lo bastante simple para poder abarcar prácticamente cualquier dominio, pero ofrece la ventaja de poder reaprovechar modelos externos y extenderlos. Además, cada

```
@Override
public void elementAdded(ContextEvent event) {
  observedAttribute = event.getAttributeInstanceArg();

  // Observe the higher context element
  observedAttribute.subscribe(this);

  // Request the values of lower context elements
  a1 = contextBroker.request(MyActuator.class,
    /* ID of the actuator */);
  a1att = a1.requestAttribute(MyActuatorAttribute.class);

  a2 = contextBroker.request(MyOtherActuator.class,
    /* ID of the other actuator */);
  a2att = a2.requestAttribute(MyOtherActuatorAttribute.class);

  // ...
}

@Override
public void elementUpdated(ContextEvent event) {
  observedValue = event.getAttributeInstanceArg().getValue();

  // Recalculate the actuators values
  // depending on the new higher value
  // ...
  a1att.update(new DefaultContextValue(value1, 1.0));
  a2att.update(new DefaultContextValue(value2, 1.0));
  // ...
}

@Override
public void elementRemoved(ContextEvent event) {
  contextBroker.release(a1);
  contextBroker.release(a2);
  // ...
  observedAttribute.unsubscribe(this);
}
```

Listado 6.8: Código para impementar una evaluación *top-down*.

actualización del contexto es interceptada por el *contenedor del contexto*, que puede decidir qué valores tomar dependiendo de criterios que se especifican en la capa superior.

El procesamiento del contexto se lleva a cabo mediante evaluaciones *perezosas*, es decir, un componente dedicado a hallar un tipo de contexto sólo se activa si otro ha solicitado previamente su valor. Lo mismo ocurre si otro componente se dedica a realizar acciones conforme a cambios en el contexto. Este componente sólo observa estos cambios si alguien ha declarado efectivamente que va a hacerlos. De este modo, sólo los recursos que sean necesarios en cada momento están activos. Puede ocurrir que en un momento dado haya varios *observadores del contexto* trabajando con los mismos *elementos de contexto*. La resolución de sus posibles conflictos se define en el siguiente capítulo, donde se establecen una serie de patrones de adaptación del sistema al contexto.

Capítulo 7

Patrones de control y adaptación

Cuando el oponente se expande, yo me contraigo. Cuando él se contrae, yo me expando. Y cuando hay una oportunidad, yo no golpeo; el golpe se da por sí mismo.

Bruce Lee.

Resumen: Este capítulo explica cómo se facilita la adaptación al contexto por parte de las aplicaciones, haciendo uso de la arquitectura. En primer lugar se explica en qué consiste la adaptación y el comportamiento oportunista, y luego se describen los comportamientos oportunistas a nivel de infraestructura y a nivel de aplicación.

7.1. Introducción

Uno de los objetivos principales de la IAm, como se menciona en el Capítulo 1, es conseguir aplicaciones que se adapten a las necesidades de los usuarios, teniendo en cuenta su *contexto*. En qué se traduce esta adaptación concretamente es un asunto de la aplicación en desarrollo. Sin embargo, en todos los casos se requieren mecanismos para controlar la elección y ejecución de los comportamientos de adaptación cuando sea necesario, de forma *oportunista* (i.e., el sistema debe esperar a las condiciones adecuadas para actuar). Esto se explica en la Sección 7.2. Además, existen formas de usar este oportunismo para plantear comportamientos adaptativos "genéricos", que cubren aspectos que pueden ser comunes a distintas aplicaciones, en forma de flujos de trabajo. Esto se explica en la Sección 7.3.

7.2. Control adaptativo oportunista

El comportamiento oportunista (Patalano y Seifert, 1997) consiste en ser capaz de ejecutar tareas cuando el contexto cumple ciertas condiciones, y hacerlo de una forma adaptada a estas condiciones. Este comportamiento necesita mecanismos de control para evaluar las condiciones, y para suspender y reiniciar tareas dependiendo de estos cambios. Este tipo de control es relevante para las aplicaciones IAm por sus cambios dinámicos y naturaleza móvil. Esto es el resultado del uso de tecnologías tan comunes a día de hoy como la localización geográfica o los dispositivos "*plug and play*".

Hay distintos tipos de control oportunista en aplicaciones IAm, tal como se explica en la Sección 2.3.2. Al nivel de aplicación se hace planificación y flujos de trabajo oportunistas. Al nivel de infraestructura hay redes, computación y fusión de información oportunistas.

FAERIE lleva a cabo acciones oportunistas independientes de las aplicaciones concretas, por ejemplo, explotar canales de información redundantes para ser más tolerante a fallos. Estas acciones oportunistas dependen de políticas o preferencias definidas por el administrador del sistema. Estas preferencias establecen prioridades o filtros considerando métricas obtenidas monitorizando observadores. Ejemplos de estas métricas son: disponibilidad, precisión, frescura, gasto de energía, coste de reinicio y estabilidad. Estas acciones genéricas se consideran a dos de los niveles mencionados: redes y computación.

El nivel de red incluye los mecanismos de control necesarios para interconectar componentes distribuidos en una red inestable. Esto implica que los nodos intermedios de comunicación son capaces de almacenar los mensajes recibidos y enviarlos cuando las direcciones de destino se hagan disponibles. Estos mecanismos en FAERIE se construyen usando tecnologías de conexión comunes como UPnP (Jeronimo y Weast, 2003) y Zeroconf (Steinberg y Cheshire, 2005).

Una visión más abstracta de oportunismo aparece a nivel de computación. Ésta implica el uso de recursos abstractos por parte de las aplicaciones. De este modo la infraestructura es capaz de soportar cambios en los dispositivos concretos conectados al sistema siempre y cuando sirvan para los mismos propósitos. La infraestructura a este nivel considera los tipos de eventos relacionados con cambios en la topología. Esto es necesario para reconfigurar las fuentes y objetivos de información, a fin de ser capaz de proporcionar continuidad en la provisión de servicios. FAERIE considera los siguientes tipos de cambios de contexto relativos a la topología:

- **Agregación de observadores de contexto al sistema.** Las posibles causas son:
  - *Conexión:* un nuevo observador de contexto se ha conectado al

sistema.

- *Accesibilidad:* un observador de contexto en un entorno remoto se hace accesible, incluyendo sus capacidades para calcular nuevos elementos de información de contexto.
- *Recuperación:* un observador de contexto que no estaba funcionando correctamente ha recuperado su funcionamiento normal.

- **Eliminación de observadores de contexto del sistema.** Las posibles causas son:
  - *Desconexión:* un observador de contexto es desconectado manualmente del sistema.
  - *Inaccesible:* un observador de contexto en un entorno remoto deja de ser accesible.
  - *Error:* un observador de contexto deja de funcionar por un error.
- **Cambio de prioridad, preferencias de filtrado o métricas.**
  - *Cambio de métricas:* las métricas monitorizadas del observador de contexto devuelven valores que no coinciden con las preferencias del sistema.
  - *Cambio de preferencias:* la modificación de las preferencias del sistema hacen inválidas las actualizaciones de ciertos observadores de contexto.

Estos tipos de cambios de contexto son gestionados por observadores específicos llamados *observadores del contexto computacional*. Estos tienen ciertos privilegios para adaptar el comportamiento del sistema conforme cambia el entorno. Sus privilegios consisten en la capacidad para manipular directamente los flujos de información, mediante la activación o desactivación de otros observadores de contexto. Esto es necesario para mejorar el rendimiento del sistema de una forma oportunista, aprovechando las distintas posibilidades del sistema. Las Figuras 7.1 y 7.2 ilustran respectivamente los componentes y el comportamiento relacionado con uno de estos observadores.

La Figura 7.1 muestra el *observador de alternativa* (*alternative observer*), que es un observador del contexto computacional. Un elemento gestor del contexto (*context manager*) crea este *observador de alternativa* para cada elemento de contexto existente. El observador lleva a cabo las actividades que se muestran en la Figura 7.2. Monitoriza ciertas métricas en el contexto para seleccionar entre diferentes observadores aquel que actualizará el elemento de contexto. Para esto, mantiene una lista de observadores de contexto candidatos a manipular el elemento de contexto, y reconsidera esta lista periódicamente. Al final, el candidato seleccionado es el que queda mejor puntuado en las métricas seleccionadas para dar prioridad.

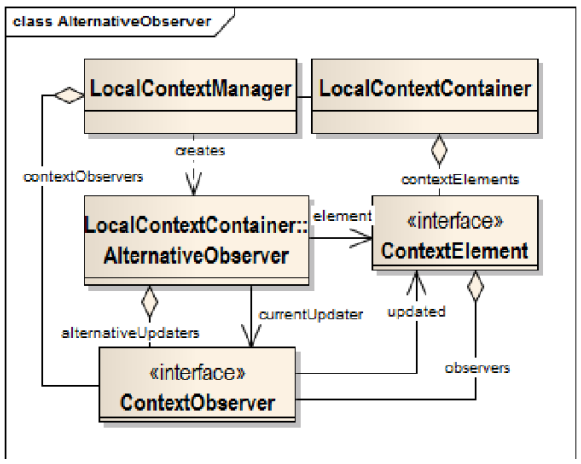

Figura 7.1: Relaciones del observador de alternativa.

La lista de observadores de contexto candidatos puede adoptar diferentes políticas:

- **Candidato único:** sólo un observador de contexto queda activo tras leer las métricas. Típicamente es el preferido de acuerdo a las métricas recogidas.
- **N candidatos:** un cierto número de candidatos quedan activos (produciendo valores de contexto). Esto produce información redundante, que tiene que ser gestionada.
- **Todos son candidatos:** usando esta política no es necesaria reconsideración. Todos los observadores se mantienen activos y se hace la necesaria gestión de la redundancia. De esta política se espera que produzca un mayor consumo de recursos, pero puede tener otras ventajas, como un menor tiempo de respuesta.

Mantener una lista de candidatos activos para un elemento de contexto fuerza a la infraestructura a tratar con información redundante. Hay diferentes políticas posibles para tratar con este asunto:

- **Todas las actualizaciones son válidas:** todas las actualizaciones hechas por observadores se aceptan, siempre que vengan ordenadas temporalmente.
- **Lista negra/Lista blanca:** todas las actualizaciones hechas por observadores de la lista blanca se aceptan, mientras que aquellas hechas por miembros de la lista negra se rechazan.

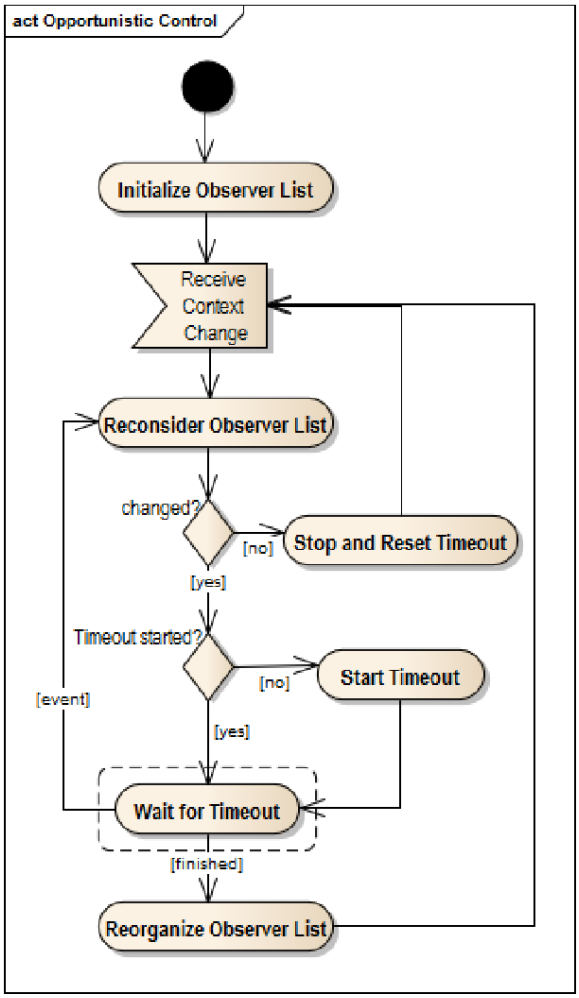

Figura 7.2: Diagrama de actividades del observador de alternativa.

- **Filtrado por métricas:** sólo las actualizaciones hechas por observadores que cumplan las métricas de preferencias especificadas se aceptan, por ejemplo, filtrar los valores con una precisión más baja que un valor dado.

- **Priorizar por métricas:** si se reciben múltiples actualizaciones en la misma ventana temporal, aquellas con valores de métricas preferibles se eligen (por ejemplo, aquellas con mayor precisión).

- **Limitar la frecuencia de actualización:** el número de actualizaciones por ventana temporal se limita, y todas las actualizaciones extra se descartan.

En resumen, FAERIE proporciona mecanismos y políticas concretos para tratar con los escenarios oportunistas identificados, que se aplican a las capas más bajas de abstracción del sistema. Estos mecanismos gestionan los flujos de información del sistema, monitorizando los posibles recursos alternativos y los cambios de contexto, reajustándolos a las nuevas circunstancias y posiblemente disparando comportamientos alternativos.

7.3. Comportamiento oportunista a nivel de aplicación

FAERIE soporta oportunismo en capas de nivel de abstracción más altas. Estas capas se caracterizan por componentes que consideran condiciones de contexto y comportamientos complejos.

Las aplicaciones del framework se construyen alrededor del concepto de *flujo de trabajo*. Un flujo de trabajo define sus actores, recursos y actividades, así como las transiciones condicionales que conectan estas actividades. Las actividades pueden tener lugar en varios entornos FAERIE. El uso de flujos de trabajo soporta la definición abstracta de actividades que serán instanciadas en distintas actividades en ejecución dependiendo del contexto real. Por ejemplo, permite ejecutar tareas cuando ciertos recursos están disponibles, o cuando la localización cambia.

FAERIE aplica mecanismos de control oportunista a este tipo de flujo de trabajo. En una aplicación en ejecución, los *observadores del contexto* monitorizan los cambios en las condiciones del mismo para disparar la instanciación de flujos de trabajo o la transición entre actividades de flujos existentes. Las transiciones disparadas pueden implicar acciones sobre el entorno, y evaluaciones y procesos internos al sistema. De este modo, el sistema puede reaccionar de una forma oportunista a la evolución de su entorno.

7.3.1. Flujos de trabajo

Los flujos de trabajo se describen como redes de actividades en diagramas de actividades. La gestión de un flujo de trabajo implica mantener su estado, y leer o manipular el contexto cuando sea necesario mientras se ejecutan las actividades en el orden establecido. Esto se hace llevando a cabo evaluaciones *bottom-up* o *top-down* respectivamente. De este modo, si la terminación de una actividad en un flujo de trabajo se determina por cierta condición del contexto (por ejemplo, termina cuando el atributo *posición* de una entidad *usuario* toma un valor *entrada-A*), se haría una evaluación *bottom-up* para examinar el valor del elemento correspondiente. De forma similar, si otra actividad implica efectuar cierta acción abstracta, (por ejemplo, transmitir un mensaje a los usuarios del entorno), se hace una evaluación *top-down* para modificar el atributo de la entidad correspondiente del entorno.

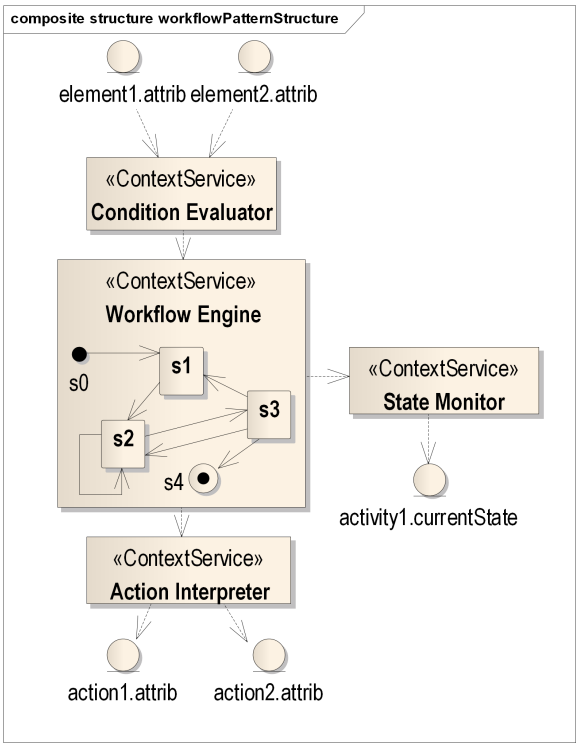

Figura 7.3: Componentes que intervienen la gestión de los flujos de trabajo.

La arquitectura soporta la funcionalidad previa con el patrón de flujo de trabajo de la Figura 7.3. Comprende los siguientes elementos:

- **Evaluador de condiciones** (`Condition Evaluator`). El patrón define un flujo de trabajo como un diagrama cuyas actividades pueden leer el contexto. Este componente pregunta al *contenedor del contexto*, generando una evaluación *bottom-up*. Entonces, los resultados de esta evaluación se usan en el motor de flujo de trabajo para resolver nodos de decisión o generar señales. Actúa como un adaptador para eliminar la gestión de contexto de la definición del flujo de trabajo.

- **Motor de flujo de trabajo** (`Workflow Engine`). Este componente almacena la definición abstracta del flujo de trabajo y gestiona su estado actual. Recibe información del *evaluador de condiciones* para determinar si cierta actividad se ha completado, o para elegir las transiciones correctas en los nodos de decisión. También, de acuerdo a la actividad a ejecutar, determina la salida a enviar al *intérprete de acciones*.

- **Intérprete de acciones** (`Action Interpreter`). Recibe acciones abstractas del *motor de flujo de trabajo*. Estas acciones indican cambios

en el contexto para ser redirigidos. Estos cambios se llevan a cabo como evaluaciones *top-down*, de modo que el resto de los componentes interesados en el sistema sean notificados.

- **Monitor de estado (`State Monitor`)**. Los mismos flujos de trabajo pueden verse como una *entidad* de la cual su "estadoActual" (*currentState*) sea interesante para ciertos componentes. El "estadoActual" es una relación que lo asocia con la actividad en curso. Este componente monitoriza el estado del flujo de trabajo preguntando al *motor de flujo de trabajo*, y es capaz de proporcionar la *Relationship* mencionada si se solicita. Esto permite a posibles componentes clientes conocer el estado actual del flujo de trabajo.

Esta estructura promociona aislar en distintos componentes el vocabulario del dominio del contexto y el de la lógica del flujo de trabajo, facilitando la reutilización. También proporciona formas de implementar las acciones y las evaluaciones de condiciones como elementos sensibles al contexto. Finalmente, este patrón es la base para soportar flujos de trabajo anidados (una actividad cuya implementación es otro flujo de trabajo) e instanciación de flujos de trabajo (una actividad que crea una instancia de otro flujo de trabajo independiente).

Las actividades anidadas ofrecen la posibilidad de descomponer una actividad más aún. Esto ocurre cuando el componente que determina la conclusión de una cierta actividad implementa a su vez otro patrón de flujo de trabajo, de modo que la conclusión depende de la finalización del segundo flujo de trabajo. Estas actividades anidadas permiten extender y adaptar la funcionalidad del sistema de una forma dinámica sin reiniciarlo.

La instanciación de flujos de trabajo se hace de la misma forma que su anidamiento, pero la acción causante termina cuando el nuevo flujo de trabajo es creado, en lugar de cuando finaliza. Creando instancias de flujos de trabajo con la lógica apropiada, varios actores pueden interactuar con el sistema IAm a la vez. Éstos pueden participar en el mismo tipo de flujo de trabajo al mismo tiempo, y el sistema mantendrá una instancia distinta por cada actor. Para este propósito se puede crear un flujo de trabajo cuyo objetivo es detectar las condiciones en las cuales hay que iniciar otra instancia de flujo de trabajo distinta. El flujo de trabajo que hace esta tarea se llama *flujo de trabajo iniciador*.

7.4. Conclusiones

El comportamiento adaptativo en un SSC puede surgir a dos niveles: a nivel genérico de la infraestructura, y a nivel de aplicación. El primero trata con eventos genéricos que pueden surgir en cualquier aplicación sensible al contexto. FAERIE da una serie de políticas para tratar con estos eventos,

e implementa un elemento, el *observador de alternativa*, que se encarga de adaptar el comportamiento según corresponda. De esta manera, FAERIE cubre un conjunto de situaciones genéricas, que son aquellas relativas a gestionar la existencia de fuentes de información redundantes, y la manera de adaptar el funcionamiento del sistema para tratar con ellas de una manera eficiente.

El segundo nivel, el nivel de aplicación, trata con eventos que son específicos de cada aplicación y la manera de adaptar el comportamiento a los mismos. Para este fin, se hace uso del concepto de *flujo de trabajo sensible al contexto*, que se define como un conjunto de actividades interconectadas, cuyas acciones y condiciones de transición se establecen en función de la información del contexto. La definición de cada flujo de trabajo dependerá de la aplicación que está siendo implementada, pero FAERIE proporciona un patrón que permite incluir cualquier tipo de gestión de flujos de trabajo. Este patrón implica proporcionar componentes que traducen los cambios en el contexto en señales al motor de gestión de flujo de trabajo, y por el otro lado componentes que traducen las acciones abstractas del flujo de trabajo en cambios en el contexto. Además, indica la manera de construir flujos de trabajo "de orden superior", cuyo objetivo es gestionar la creación y destrucción de otros flujos de trabajo, permitiendo así llevar a cabo múltiples interacciones con el sistema.

Capítulo 8

Sistemas multi-agente en IAm

... cada uno debe ser puesto a un trabajo, que ha de ser aquel para el que esté dotado, ...

Platón.

Resumen: En este capítulo se propone el uso integrado de agentes software en FAERIE como participantes en flujos de trabajo dentro de SSC. En primer lugar se hace una breve introducción a la utilización de agentes en SSC, después se explica el diseño de un agente software con FAERIE y finalmente se discute el resultado.

8.1. Introducción

Los flujos de trabajo descritos en el capítulo anterior incorporan en su especificación las lógicas de interacción con el SSC de sus distintos participantes, sean humanos o componentes software. En algunos casos, el proceso de decisión de estos componentes puede incorporar multitud de detalles específicos de un dominio de aplicación o ser altamente complejos, por lo que se puede requerir un nivel de abstracción muy elevado a la hora de definirlos. Una manera adecuada para modelar el comportamiento de los componentes en estos casos es utilizando el concepto de *agente software*. Un agente software es una entidad de modelado con características sociales e intencionales, que trabaja principalmente en términos de transformaciones de información y de colaboración con otros agentes (Weiss, 2000).

La utilización de este concepto para modelar participantes en flujos de trabajo es interesante, porque existen múltiples tareas complejas estudiadas para este modelo que pueden aplicarse a los problemas de IAm. Estas tareas incluyen gestión de diálogo, negociación, colaboración, aprendizaje

automático, planificación y toma de decisiones. Además, el desarrollo puede beneficiarse del uso de metodologías de diseño de agentes existentes, como INGENIAS (INGENIería para Agentes Software) (Gómez-Sanz et al., 2010). Por ejemplo, el proyecto ROSACE trata una serie de protocolos de negociación entre agentes para elegir el uso más adecuado de los recursos en un contexto de incendio forestal, y ha sido implementado con Ícaro (Lacouture et al., 2012). Para FAERIE se propone que ciertas tareas muy complejas sean definidas por medio de un SMA (Sistema Multi-Agente), mientras que las tareas menos complejas sean manejadas por *observadores de contexto* FAERIE básicos.

A continuación se presenta una solución de integración para agentes software dentro de la infraestructura FAERIE. Esta integración se hace de tal manera que los agentes puedan también beneficiarse de las facilidades de procesamiento del contexto de la plataforma. Una de las ventajas que adquieren es el acceso a una fuente de información compartida y genérica (el contexto), de modo que requerirán menos interacciones para obtener cierta información.

8.2. Diseño e integración de agentes sensibles al contexto

Existen múltiples infraestructuras para implementar agentes software (p.ej., JADE (*Java Agent DEvelopment Framework*) (Bellifemine et al., 2001), Ícaro (Gascueña et al., 2010) o JAF (*Java Agent Framework*) (Vincent et al., 2001)). El único requisito para integrar estos agentes en FAERIE es que gestionen un componente FAERIE que proporcione la interfaz `ContextObserver`, tal y como se describe en el Capítulo 5. Esto les permite observar y cambiar la información de contexto, y beneficiarse de las capacidades de la infraestructura para tratar con recursos cambiantes, tal y como se describe en los Capítulos 6 y 7. Esta integración se situaría por encima de las capas descritas anteriormente (ver Capítulo 4). Este acceso a un modelo abstracto de información permite diseñar el agente de una forma más simple, debido a que la plataforma ya gestiona parte del tratamiento de la misma.

Si bien lo anterior es lo único necesario para que los agentes hagan uso de la arquitectura, a continuación se propone un paso más para mejorar el uso de la plataforma por parte de los agentes. Como en muchas teorías de agentes, en esta propuesta los agentes individuales poseen su propio "estado mental". Este estado mental actúa para el agente como un contenedor de contexto privado donde maneja su propia información. De este modo se alinea la gestión de este estado mental con la del resto del contexto en los *contenedores de contexto*, de tal manera que se considera la existencia de un *contenedor de contexto* localizado internamente en el agente, tal como muestra la Figura 8.1. Este *contenedor de estado mental* actúa como el *contenedor de contexto*

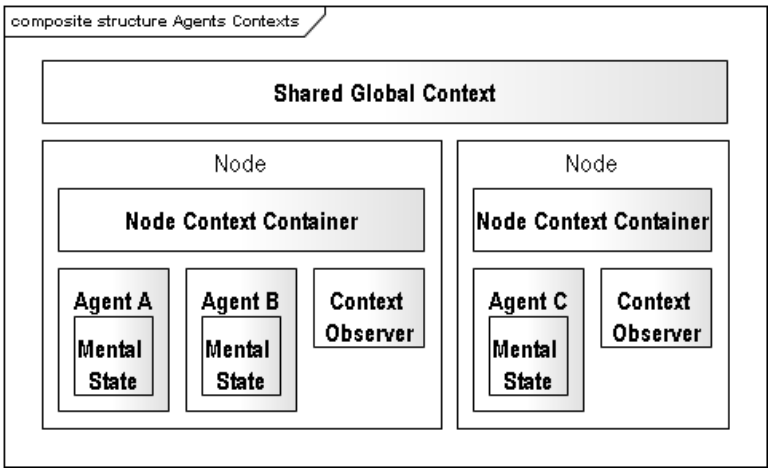

Figura 8.1: Organización de los niveles de contexto en FAERIE.

descrito en capítulos anteriores. Cada vez que un agente crea una entidad para actualizar la representación de su estado propio y del entorno, puede especificar si la entidad se crea públicamente o de forma privada. Cuando hace esto, el agente comprobará si él mismo es capaz de resolver la petición, y si no, ésta será redirigida al *contenedor de contexto* de su nodo. Este mecanismo funciona del mismo modo que cuando el contenedor de contexto local pide información a un entorno remoto. Esto permite prevenir que el *contenedor de contexto* almacene información que no es interesante o pública para otros componentes del nodo. Por otra parte, la existencia de un *contenedor de contexto* accesible para distintos agentes reduce el intercambio de mensajes para informar a otros agentes, ya que ya tienen accesible la mayoría de la información básica del *entorno* actual. La capa superior de la Figura 8.1 representa un contenedor abstracto global. Es la parte del contexto de distinto nodos que comparten con otros nodos del sistema distribuido. Este contenedor no existe físicamente, ya que es el resultado de las acciones de compartición de información entre nodos.

De esta manera, se crean tres niveles de contexto:

1. **Contexto del Agente**: este contexto incluye únicamente los elementos que son interesantes para el agente en su razonamiento interno.

2. **Contexto del Nodo**: este contexto incluye los elementos necesarios para la funcionalidad del entorno FAERIE.

3. **Contexto del SSC**: este contexto incluye los elementos que son compartidos entre distintos entornos FAERIE.

El diseño y estado mental de los agentes se representan utilizando concep-

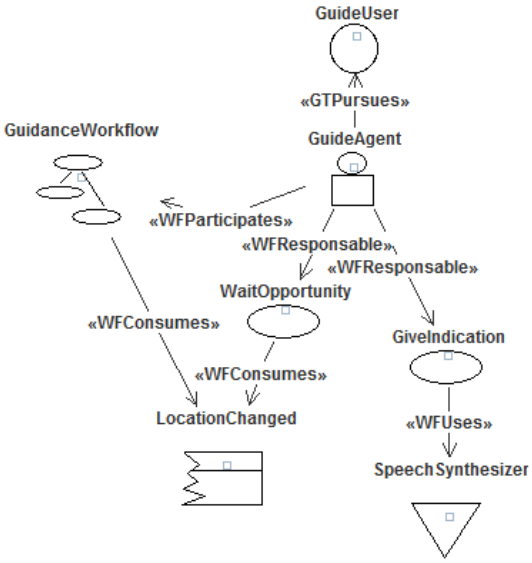

Figura 8.2: Ejemplo de definición de un agente utilizando la notación INGENIAS.

tos comunes del paradigma de agentes. La Figura 8.2 muestra un diagrama del caso de estudio de la Sección 11.1 siguiendo la notación INGENIAS. La definición del *GuideAgent* considera los *objetivos* que persigue (círculos), las *tareas* que puede llevar a cabo (elipses) y los *flujos de trabajo* donde participa para alcanzar esos objetivos (símbolo de diagrama de actividades). Las tareas producen y consumen partes de información (e.g., *eventos* y *hechos*, los rectángulos recortados), y hacen uso de recursos o aplicaciones (triángulos). Para llevar a cabo la integración, cada uno de estos conceptos se alinea con los elementos del modelo de contexto utilizado. Por ejemplo, `SpeechSynthesizer` sería el nombre de un tipo de entidad en el modelo de contexto que es modificada y `LocationChanged` sería otro tipo de elemento de contexto que sería observado.

En el caso de INGENIAS (Gómez-Sanz et al., 2010), el kit de desarrollo se usa para el modelado y generación de código, y la infraestructura proporciona las bibliotecas básicas para el código del agente y herramientas de depuración. Esta generación de código puede personalizarse para hacer uso de las librerías de FAERIE.

8.3. Conclusiones

En este capítulo se ha discutido la conveniencia de utilizar SMA para desarrollar aplicaciones *sensibles al contexto* complejas. Una de sus principales

ventajas es el nivel de abstracción de este paradigma. Gracias a esto, pueden definirse comportamientos altamente complejos que aborden un gran número de tareas. La integración con la infraestructura de FAERIE presentada con anterioridad es sencilla. Consiste meramente en hacer que los agentes proporcionen, quizás a través de algún recurso que gestionen, la interfaz `ContextObserver`, especificada en el Capítulo 6, lo que les da acceso a los servicios de la infraestructura.

Además de esta solución inmediata, se propone otro nivel de integración, que consiste en tratar la memoria de trabajo de los agentes como si fuese otro nivel de *contenedor del contexto*. Esto permite extender los mecanismos de gestión de contexto a los agentes y separar la información. Ello facilita el control de la información que puede considerarse privada al agente y también ahorra al *contenedor del contexto* del nodo almacenar información que sólo es útil para un único agente.

Capítulo 9

Desarrollo de un sistema FAERIE

Para hacer una tortilla tienes que romper algunos huevos.

Tyler Durden.
El club de la lucha.

Resumen: En este capítulo se presentan las cuestiones relativas al proceso de desarrollo de un sistema FAERIE. En primer lugar, se explica un proceso que sirve de guía para implementar aplicaciones siguiendo la arquitectura, desde la fase de obtención de requisitos hasta la de despliegue y pruebas. En segundo lugar, se describen los mecanismos de la infraestructura para facilitar el proceso de construcción y despliegue de una aplicación FAERIE.

9.1. El proceso de desarrollo

El proceso propuesto para diseñar un SSC en FAERIE se divide en estas fases: *consideraciones*, *identificación*, *modelado*, *implementación*, *despliegue* y *pruebas*. El proceso se lleva a cabo de una forma iterativa e incremental, y cada una de sus fases se describe a continuación. Un ejemplo de su aplicación puede verse en la Sección 11.1.

9.1.1. Consideraciones

La fase de *consideraciones* es una etapa preliminar en la cual se estudian las características de los *entornos* y las entidades involucradas en la

aplicación. El objetivo es relacionar los requisitos de la aplicación con las condiciones reales del escenario de despliegue. Las condiciones incluyen:

- *Características de los entornos físicos.* Por ejemplo, el uso de batería y la capacidad computacional de un entorno móvil, el tamaño y distribución de espacios, y el nivel de ruido y características de un acceso a la red.
- *Componentes disponibles y capacidades.* Algunos ejemplos a considerar son los componentes físicos (e.g., periféricos, routers y sensores), componentes software (e.g., controladores e intérpretes), y servicios externos (e.g., servicios web de mapas, y componentes para acceso a bases de datos remotas).
- *Caracteristicas de los usuarios.* Los factores principales son el número de usuarios y sus preferencias, incluyendo asuntos de privacidad y necesidades especiales si existen.

El resultado es un texto más o menos estructurado remarcando todas las consideraciones observadas, y los elementos más importantes que puedan estar involucrados.

9.1.2. Identificación

La fase de *identificación* determina los elementos clave de la aplicación basándose en el análisis de la fase de *consideraciones*. El resultado es una lista de *elementos de contexto*, y las reglas que influyen en sus cambios.

9.1.2.1. Elementos

La fase de *consideraciones* permite identificar algunos elementos que se espera sean relevantes para el funcionamiento de la aplicación. Éstos van a constituir parte del *contexto*, y serán representados en FAERIE como *elementos del contexto* (ver Sección 6.2).

De aquí se obtiene una lista, extraída del análisis de la descripción de los requisitos y de las consideraciones. Esta lista puede aumentar o disminuir en sucesivas iteraciones del proceso. En esta fase, todo el vocabulario del sistema es simplemente reunido para tomarlo en consideración. Estos elementos son refinados posteriormente a sus representaciones estructuradas en la fase de *modelado*.

9.1.2.2. Reglas

El diseño de la aplicación también requiere determinar las dependencias entre los elementos de contexto. Estas dependencias guían el desarrollo, ya

que la aplicación necesita componentes capaces de resolverlas. Establecen la estructura de la fusión de información a llevar a cabo, como se discute en la Sección 6.3. Ésta puede ser de los tipos: agregación, derivación y adaptación. El objetivo de esta fase es obtener un conjunto de reglas que establezcan el estado de cada uno de los elementos en función del estado de otros elementos. El nivel de formalidad y detalle de esta especificación irá aumentando en sucesivas iteraciones del proceso.

9.1.3. Modelado

La fase de *modelado* consiste en representar los elementos y reglas identificados en la fase anterior como un modelo de contexto. Este modelo de contexto utiliza los *elementos de contexto* descritos en la Sección 6.2. Si las entidades, relaciones y atributos han sido identificadas correctamente, el modelado consiste en representarlas en un diagrama de clases.

Dado que la arquitectura permite definir modelos que extienden otros modelos, es conveniente intentar alinear el vocabulario para hacer uso de modelos de contexto existentes, mejorando así la reutilización de sus componentes de procesamiento. Para conseguir esto, sin embargo, ha de cumplirse que la semántica asociada a este vocabulario sea exactamente la misma, de tal modo que dos *observadores de contexto* que utilicen el mismo tipo de elemento del contexto no tengan conflicto en interpretar qué significa.

9.1.4. Implementación

La fase de *implementación* produce los componentes que ejecutan las reglas obtenidas en la fase de *identificación*, aplicadas sobre el modelo de contexto desarrollado en la fase de *modelado*. Cada uno de estos componentes será un *observador del contexto*. En esta fase, los desarrolladores necesitan distinguir entre componentes que pueden reutilizarse de otras aplicaciones y los componentes que han de desarrollarse desde cero.

Cuando el modelo de contexto extiende uno existente, pueden existir *observadores de contexto* existentes para alguno de los elementos. Si estos observadores cumplen los requisitos de la nueva aplicación, pueden utilizarse dentro del *conjunto de despliegue* como un componente externo. Por ejemplo, puede existir un componente que es capaz de obtener el *Tiempo de respuesta* para los *Enlaces Wi-Fi*. Sin embargo, los ingenieros pueden decidir desarrollar nuevos *observadores de contexto*, ofreciendo implementaciones alternativas. Esta redundancia será manejada por la plataforma.

9.1.5. Despliegue y pruebas

El despliegue del sistema se hace generando una configuración de despliegue para cada tipo de entorno identificado. Esto generará un conjunto

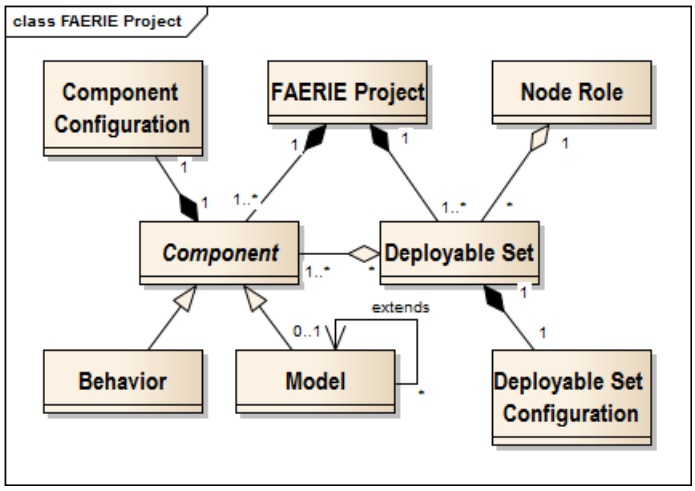

Figura 9.1: Estructura de un proyecto FAERIE.

desplegable para cada tipo de entorno. Lo siguiente es ejecutar los conjuntos desplegables, tal y como se especifica en la Sección 9.2. Si el despliegue es en un entorno virtual, todos los nodos serán probablemente ejecutados dentro de la misma máquina (como distintos procesos).

9.2. Mecanismos de desarrollo

Esta sección es una introducción a cómo se trabaja con FAERIE para crear un proyecto de SSC. Teniendo en cuenta la concepción de SSC explicada anteriormente, se va a describir qué forma tiene un proyecto FAERIE, y cómo se construye para dar lugar a un SSC.

9.2.1. Estructura de un proyecto FAERIE

La Figura 9.1 muestra los elementos que componen un proyecto FAERIE. Los elementos principales son los *componentes* (*component*), los cuales pueden ser de *modelo* (*model*), o de *comportamiento* (*behavior*). Los componentes de modelo aportan nuevos tipos de elementos de contexto para manipular, y pueden extenderse unos a otros, representando nuevos dominios de negocio. Los componentes de comportamiento aportan procesos, los cuales normalmente manipulan los elementos proporcionados por los componentes de modelo. Estos componentes están separados en distintos *conjuntos desplegables* (*deployable sets*). Cada conjunto desplegable es un conjunto de componentes que debe desplegarse en un dispositivo principal diferente. El tipo de dispositivo es representado por un *rol de nodo* (*node role*). Ejemplo de roles de nodo son "dispositivo móvil del usuario" o "PC del museo".

Tanto los componentes como los conjuntos desplegables tienen sus propias configuraciones, que incluyen un conjunto de ficheros necesarios para su funcionamiento. Estos ficheros incluyen configuración independiente de la plataforma, y dependiente de la plataforma, y especifican parámetros de arranque, como la cantidad de memoria disponible o la configuración del registro (*log*). El proceso de compilación y empaquetado dependiente de la plataforma forma parte del proceso de construcción, y se explica en la siguiente sección.

9.2.2. Proceso de construcción

El proceso de construcción de un proyecto FAERIE descrito a continuación trata de cumplir con los requisitos de desarrollo identificados I., II., III., IV. y V. La herramienta utilizada para facilitar este proceso es Apache Maven (Massol y O'Brien, 2005). Maven mantiene un repositorio local y uno remoto para almacenar los componentes que genera. Además, permite establecer dependencias entre ellos, permitiendo hacer uso de los servicios de unos en la implementación de los otros.

El resumen del proceso es el siguiente:

1. Se crea un conjunto de módulos con estructura y contenido por defecto haciendo uso del arquetipo *FAERIE Project*. Un arquetipo es una especificación de la estructura de un proyecto con múltiples módulos, que se instancia especificando los identificadores de los módulos a considerar.

2. Se especifican las dependencias de los módulos entre ellos, posiblemente incluyendo módulos externos. También se especifica cómo se reparten éstos entre los distintos conjuntos desplegables a generar.

3. Se crean los tests unitarios para cada uno de los módulos.

4. Se crean los tests de integración que hacen pruebas de módulos combinados.

Tras la creación del proyecto, comienza el ciclo de desarrollo con integración continua. Consta de los siguientes pasos, que se ejecutan en secuencia, de tal modo que sólo se continúa con el siguiente si el anterior no ha producido ningún fallo:

1. Codificar un fragmento de la aplicación.

2. Ejecutar tests unitarios.

3. Ejecutar tests de integración locales.

4. Desplegar en el servidor de integración continua.

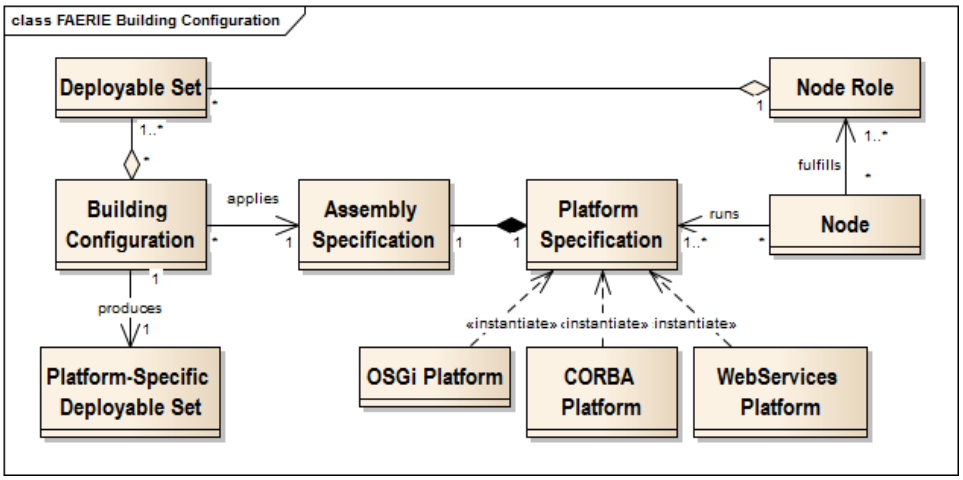

Figura 9.2: Proceso de adaptación a la plataforma.

Una vez desplegado en el servidor de integración continua, éste vuelve a ejecutar de nuevo todas las pruebas, pero esta vez con sus versiones de los módulos. En este caso, las pruebas pueden no resultar exitosas, debido a los cambios potenciales de los módulos por parte de otros miembros del equipo de desarrollo. Si este es el caso, el servidor notifica a los desarrolladores para que traten de integrar los distintos cambios en la siguiente iteración.

El último aspecto a tratar es la adaptación a la plataforma de ejecución a la hora de llevar a cabo las pruebas. Las diferencias fundamentales en la ejecución en distintas plataformas dependen de dos cuestiones: de la interfaz de acceso de las mismas, y de la estructura que ha de tener los componentes que se ejecutan. La Figura 9.2 muestra cómo un conjunto desplegable independiente se transforma en uno dependiente de plataforma mediante un proceso de construcción determinado. Este proceso hace uso de una *especificación de ensamblaje* (*assembly specification*). Este concepto pertenece a Maven, y representa la especificación de una estructura de ficheros y un método de compresión. Cada *plataforma* (*platform*) tiene una especificación de ensamblaje diferente, y Maven se encarga de que cada conjunto desplegable cumpla esta especificación según la plataforma elegida, mediante la *configuración de construcción* (*building configuration*). Esta plataforma se representa con el parámetro *perfil* a la hora de ejecutar Maven. El resultado es un *conjunto desplegable específico de la plataforma* (*platform-specific deployable set*).

La adaptación de la capa de infraestructura a la API de cada plataforma la lleva a cabo el *bróker del contexto*, como se explicó en el Capítulo 5.

9.3. Conclusiones

El proceso utilizado ofrece un buen equilibrio entre simplicidad y generalidad. Ofrece un enfoque de desarrollo más simple que otras alternativas más formales y generales, tales como el trabajo de Sánchez-Pi et al. (2012), y proporciona más guía y correlación con las herramientas de desarrollo proporcionadas.

El capítulo describe cómo es la estructura de un proyecto FAERIE. Un proyecto está compuesto de una serie de *conjuntos desplegables*, que especifican unos componentes que han de ser desplegados en unos ciertos nodos (especificados de forma genérica). En el proceso de construcción se especifica a qué plataforma concreta se va a desplegar para agregar los elementos necesarios. De este modo, mucha de la complejidad relativa a esta parte del desarrollo queda cubierta por la infraestructura de desarrollo. El proceso de construcción, además, tiene en cuenta el despliegue automático en diversos entornos, lo que facilitará crear un entorno de simulación, como se verá en el siguiente capítulo. También incluye la ejecución de pruebas sobre el sistema, facilitando así adoptar estas prácticas de desarrollo.

Capítulo 10

Integración con escenario virtual

¿Sabes? Sé que este filete no existe, sé que cuando me lo meto en la boca, es Matrix la que le está diciendo a mi cerebro: "es bueno, y jugoso". Después de nueve años, ¿sabes de qué me doy cuenta? La ignorancia es la felicidad.

Cifra hablando con el agente Smith.
Matrix.

Resumen: En este capítulo se explica la necesidad y los beneficios que aporta la simulación de escenarios en sistemas IAm. Posteriormente se describe la herramienta utilizada en FAERIE, UbikSim, y la forma en que se utiliza con la infraestructura.

10.1. Simulación en sistemas de IAm

La validación de los requisitos de aplicaciones IAm puede ser costosa y representar un desafío. Muchas aplicaciones involucran potencialmente un gran número de usuarios y dispositivos, y ciertos escenarios de prueba se desarrollan en largos periodos de tiempo. Aparte, esta validación se vuelve impracticable en algunas situaciones extremas, como en un escenario de incendio o evacuaciones.

Existen múltiples proyectos de investigación que se llevan a cabo en entornos reales. A estos entornos se les denomina "*living labs*", y son ecosistemas localizados normalmente en un contexto territorial (p.ej., una ciudad o una urbanización), donde se introducen distintos tipos de tecnologías, muchas de

ellas orientadas a la IAm, para su desarrollo, prueba y validación (Bergvall-Kareborn et al., 2009). Sin embargo, el objetivo en este trabajo es poder implementar prototipos rápidamente, a bajo coste, y agilizar el proceso de desarrollo.

Para lograr estos objetivos, se ha optado por la vía de la simulación de entornos IAm virtuales (Serrano y Botía, 2013). Este enfoque permite ejecutar experimentos en un entorno muy controlado, y que permite ver inmediatamente los resultados de distintos cambios en el diseño. Estos experimentos pueden ejecutarse además "en lotes" y en tiempos más cortos que los experimentos "reales", obteniendo rápidamente los resultados. Otra característica importante es la capacidad de replicar los experimentos, ya que todo el entorno está bajo el control total de los ingenieros/desarrolladores. Esta aproximación también tiene sus limitaciones: los resultados de la simulación no serán tan "realistas" como los obtenidos en entornos reales, por lo que se necesitarán pruebas adicionales de validación en el entorno real. Sin embargo, incluir simulaciones en el desarrollo ofrece la ventaja clave de que los errores que se encuentren en las fases tempranas del desarrollo serán más fáciles y baratos de resolver. Las ventajas mencionadas tienen el peso suficiente como para abordar esta alternativa. Este capítulo hace referencia al requisito IV. identificado en el Capítulo 3.

A continuación se describe la herramienta utilizada en FAERIE para la simulación, UbikSim (Campuzano et al., 2011), y posteriormente se explica cómo se integra esta herramienta en la plataforma.

10.2. Validación y pruebas con UbikSim

UbikSim es un kit de desarrollo que proporciona las herramientas para llevar a cabo simulaciones de sistemas IAm antes de su despliegue real (Campillo-Sánchez y Botía, 2012). La herramienta está siendo aplicada actualmente en varios proyectos relacionados con IAm. Uno de ellos es Necesity (García-Valverde, Campuzano, Serrano, Villa y Botía, 2012), que tiene como objetivo reducir de forma automática y no intrusiva los tiempos de espera hasta que una persona mayor es atendida tras sufrir una caída en su casa o dentro de una residencia. Otro de ellos es Caronte (Team, 2011), que pretende aplicar las tecnologías Web 2.0 para conseguir una plataforma social que favorezca la colaboración en las situaciones de emergencia que se puedan producir en un centro geriátrico.

UbikSim define un SUT (*System Under Test*, Sistema En Prueba) como un módulo software, o un conjunto de ellos, involucrados en un entorno inteligente y cuya funcionalidad está en prueba. El comportamiento del SUT depende de sus entradas. Estas entradas son principalmente información contextual del entorno, en este caso de una simulación. Ejemplos de estas entradas son los valores dados por los sensores de distancia y peso.

La especificación de una simulación está compuesta por un escenario y una configuración. El escenario es un entorno que tiene una topología y un número de usuarios y dispositivos con distintos roles, propiedades (p.ej., número y localización de sensores), y comportamientos. La configuración establece el tiempo inicial y la semilla de la simulación. La semilla es un número que identifica una secuencia de números pseudoaleatorios que se utiliza para agregar aleatoriedad a las simulaciones. Utilizando la misma semilla se puede reproducir exactamente la misma simulación. Así, una prueba en UbikSim es equivalente a una simulación concreta definida por su especificación, porque dependiendo de las especificaciones la entrada de contexto simulada será diferente. Un servicio o aplicación IAm (es decir, un SUT) se valida correctamente cuando pasa todas las pruebas, es decir, todas las especificaciones de simulación deseadas. UbikSim no soporta la comprobación automática de los resultados de las pruebas, de modo que el desarrollador debe hacerlo revisando los estados del SUT y el contexto recibido.

Para las tareas anteriores, UbikSim incluye los siguientes módulos:

- UbikEditor, un editor 3D extensible que sirve para modelar los escenarios de las simulaciones. Proporciona facilidades para definir gráficamente los elementos de un entorno inteligente en el interior de una edificación y su distribución. Cada elemento tiene propiedades que pueden modificarse, como el nombre, rol, precisión o rango de un sensor.

- Un simulador, que está construido sobre la biblioteca de modelado basada en agentes MASON (*Multi-Agent Simulator Of Neighborhoods*) (Luke et al., 2005). El uso del enfoque basado en agentes permite modelar comportamientos humanos (p.ej., profesores en universidades y cuidadores en hospitales), sensores (p.ej., sensores de presencia, peso y movimiento), y actuadores (p.ej., control de aire acondicionado o luces), en una infraestructura común. Este simulador toma como entrada las especificaciones de simulaciones desarrolladas con el editor.

- Adaptadores a distintas plataformas sensibles al contexto. Una API permite a módulos externos recibir información de contexto del simulador. Esta facilidad es útil para probar módulos de terceros en un entorno simulado. Por ejemplo, datos simulados de sensores pueden utilizarse para probar un módulo de localización.

10.3. Integración de UbikSim en FAERIE

A continuación se describe cómo se hace uso de las herramientas de UbikSim en FAERIE. En primer lugar se indica cómo se integra el simulador en la ejecución de un SSC, y después la forma de integrar el uso de la herramienta

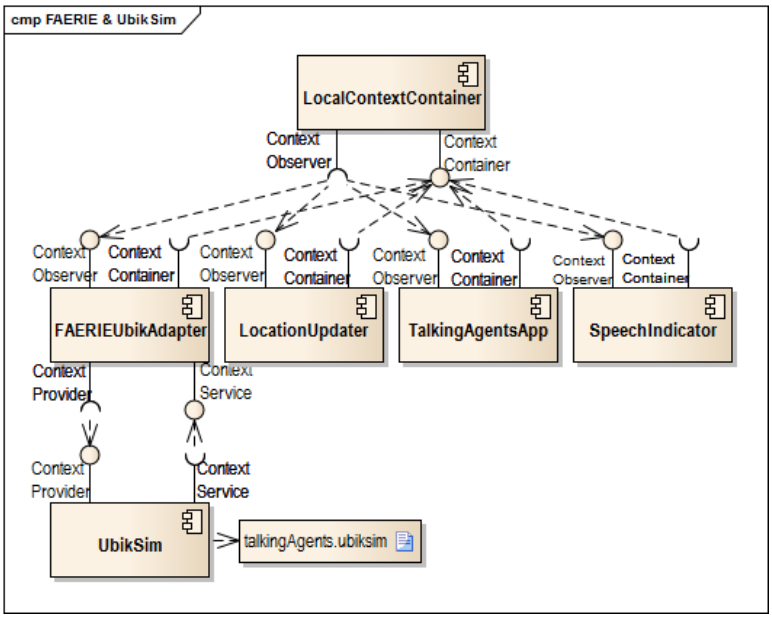

Figura 10.1: Utilización del componente UbikSim junto a otros en una aplicación FAERIE.

en el desarrollo, es decir, cómo se maneja el despliegue virtual y luego se pasa a un despliegue real.

10.3.1. Integración en la arquitectura

Al igual que se hace con los agentes descritos en el Capítulo 8, la manera de integrar el simulador de UbikSim consiste en desarrollar un adaptador que implemente la interfaz `ContextObserver` dentro de la plataforma de ejecución. La Figura 10.1 muestra un diagrama de componentes correspondiente a una aplicación FAERIE. Cada componente en la capa media es un observador de contexto conectado con el contenedor de contexto local. La integración con el componente UbikSim se hace mediante el componente *FAERIEUbikAdapter*. Éste mapea cada dispositivo en UbikSim a elementos de contexto. Después lee los eventos generados por UbikSim para actualizar los atributos de los elementos en el contexto. En este sentido, el simulador actúa como un sensor compuesto, y el adaptador es un *observador anclado*.

10.3.2. Integración en el desarrollo

Dentro del proceso de construcción descrito en el Capítulo 4, se define una *especificación de ensamblaje* para la simulación. De esta manera, se define un perfil de construcción que incluye automáticamente la librería de

UbikSim y el adaptador en el conjunto desplegable, para un determinado fichero ".ubiksim" representando un escenario. De este modo, al ejecutar la plataforma, ésta hará uso del escenario virtual en su ejecución.

Una vez se haya terminado el desarrollo y validación, se pasa al perfil de "despliegue real". En este despliegue, se sustituyen los componentes de UbikSim con los componentes que manejan los sensores reales.

10.4. Conclusiones

Este capítulo explica las dificultades que comporta llevar a cabo pruebas y validación sobre un sistema IAm en su despliegue real. Para mitigarlas, se propone el uso de simulaciones. En particular, este trabajo usa UbikSim, un kit de desarrollo que proporciona herramientas para simular entornos 3D. Esto reduce costes a la hora de realizar las pruebas iniciales del sistema.

Para cumplir uno de los requisitos iniciales especificados en el Capítulo 3, FAERIE proporciona un componente que adapta los eventos de simulación de UbikSim a cambios en el modelo del contexto de una aplicación FAERIE. Esta adaptación permite probar sistemas en entornos simulados como si se ejecutaran en un entorno real. Además, para facilitar el desarrollo de los sistemas, el despliegue en un entorno simulado se lleva a cabo de forma automática, como si se tratase de una plataforma más. Esto implica que pasar del entorno simulado al real sólo requiere especificar otro perfil de despliegue distinto, tal y como se describe en la Sección 9.2. Lo único que no se podría probar en el entorno simulado es el funcionamiento de los sensores reales, que deberían probarse mediante otros métodos.

Capítulo 11

Casos de estudio

La Arqueología busca el hecho, no la verdad. Si es la verdad lo que les interesa, el Doctor Tyree da filosofía en la clase del fondo.

Indiana Jones a sus alumnos.
Indiana Jones y la Última Cruzada.

Resumen: En este capítulo se explican los dos casos de estudio utilizados para evaluar la infraestructura de FAERIE. El primer caso de estudio incluye los resultados obtenidos en cada paso del proceso de desarrollo, para ilustrar cómo se hace uso del mismo. El segundo, más sencillo, se utiliza para mostrar uno de los patrones de control estudiados.

11.1. Escenario "Tutorías flexibles"

Este caso de estudio consiste en una aplicación que soporta un programa de tutorías flexibles en una universidad. El objetivo de la aplicación es un mejor uso del tiempo de profesores y estudiantes. Durante las horas de tutoría, los profesores se benefician de poder trabajar en tareas distintas en cualquier parte del edificio (laboratorios, despachos, salas especiales, etc.), siempre y cuando ningún estudiante requiera una reunión. Por su parte, los estudiantes se benefician de horarios de tutoría más amplios y flexibles. Este acuerdo sólo es satisfactorio si los estudiantes pueden localizar fácilmente a sus profesores cuando necesitan tutorías. Cuando esto ocurre, el sistema es responsable de inferir la localización de ambos en el edificio, y de guiar al estudiante para reunirse con el profesor.

A continuación se describen los resultados concretos de la aplicación del proceso de desarrollo descrito en el Capítulo 9.

11.1.1. Consideraciones

Componentes físicos típicos de este escenario son los *teléfonos móviles* de los *estudiantes* y *profesores*. A día de hoy, es bastante común que la gente de la universidad tenga su propio "smartphone" con una *interfaz de red*. Tales teléfonos pueden usarse para acceder a la *aplicación*, pero también para proporcionarle información de *localización* a la misma. Esta información puede obtenerse mediante triangulación de las *antenas Wi-Fi cercanas*. Por lo tanto, estas antenas son también componentes físicos de este escenario.

Usar señales inalámbricas para triangulación no es una tarea trivial. Los algoritmos involucrados deben tratar, por ejemplo, con el ruido de señal, puntos ciegos, y la agregación y retirada de antenas. Estos algoritmos están ya encapsulados y disponibles en algunos componentes software específicos, con los algoritmos de aprendizaje y adaptación requeridos.

El proceso de localización previo necesita que los profesores siempre lleven los teléfonos con ellos. Sin embargo, podrían tenerlos apagados o haberlos dejado en sus despachos. En este caso, el SSC puede recurrir a sus *horarios* públicos: si el profesor tiene una clase en el *momento actual*, entonces debería estar en el aula en la cual esta actividad tiene lugar. Esta información se encapsula para su uso en otro componente software gestionado por la *universidad*.

11.1.2. Identificación

A continuación se detallan los elementos y reglas identificados para este caso de estudio.

11.1.2.1. Elementos

Tras la lectura de las consideraciones, se han extraído los conceptos más importantes (aquellos remarcados en cursiva), obteniendo como resultado la siguiente lista:

- *Aplicación* (Entidad)
- *Universidad* (Entidad)
- *Teléfono móvil* (Entidad)
- *Estudiante* (Entidad)
- *Profesor* (Entidad)
- *Interfaz de red* (Entidad)
- *Antena Wi-Fi* (Entidad)
- *Antenas Wi-Fi en los alrededores* (Relación)
- *Tiempo de respuesta* (Atributo)
- *Localización* (Relación)
- *Horario* (Atributo)
- *Tiempo actual* (Atributo)

Tras un análisis más profundo de los elementos identificados, la lista se refina con las siguientes modificaciones:

- *Universidad* se cambia por *Lugar* dentro de la universidad. La universidad al completo no es adecuada para constituir un "espacio físico" para la aplicación. Los espacios a considerar son las localizaciones más concretas dentro del campus de la universidad.
- Las entidades *Teléfono móvil*, *Interfaz de red* y *Antena Wi-Fi* dan lugar a una entidad nueva, llamada *Enlace Wi-Fi*. Cada *Enlace Wi-Fi* es un sensor virtual que proporciona el *Tiempo de respuesta* de una única *Antena Wi-Fi*, tal como lo percibe la *Interfaz de red* de un cierto *Teléfono móvil*. De hecho, esta entidad "virtual" es la única interesante para el problema, ya que proporciona el dato utilizado para hallar la localización del usuario.
- *Antenas Wi-Fi en los alrededores* se cambia por *Sensores en los alrededores* debido a la modificación anterior.
- *Tiempo actual* se deja de considerar un `ContextElement`, ya que su cálculo no requiere ningún tratamiento especial y es accesible globalmente.

11.1.2.2. Reglas

En este caso de estudio, las dependencias entre los elementos obtenidos son:

- la *Aplicación* **observa cambios en**:
  - la *Localización* del *Profesor* Y la *Localización* del *Estudiante*. Esto es un proceso de *adaptación*.
- la *Localización* de una *Persona* **cambia con**:
  - o el *Tiempo de respuesta* de *Sensores en los alrededores* de esa *Persona*. Esto es un proceso de *agregación*.
  - o el *Tiempo actual* Y el *Horario* del *Profesor* (si la persona es un profesor). Esto es un proceso de *derivación*.

  La infraestructura manejará la redundancia de estas dos fuentes.
- los *Sensores en los alrededores* de una *Persona* **cambian con**:
  - los nuevos *Enlaces Wi-Fi* descubiertos. Esto es un proceso de *agregación*.

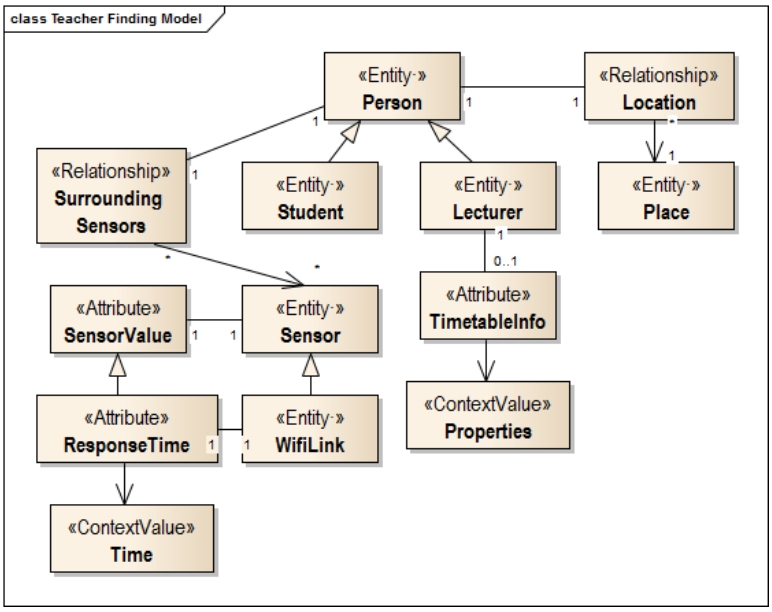

Figura 11.1: Modelo del contexto para la aplicación "Tutorías Flexibles".

- el *Tiempo de respuesta* de un *Enlace Wi-Fi* es **obtenido directamente** por un *observador anclado*.

- el *Horario* de un *Profesor* es **obtenido directamente** por un *observador anclado*.

11.1.3. Modelado

La Figura 11.1 muestra el resultado para esta aplicación.

En el caso propuesto, se hace uso de entidades ya existentes, que son las entidades *Persona* (*Person*) y *Sensor*, la relación *SensoresAlrededor* (*SurroundingSensors*), y el atributo *ValorDeSensor* (*SensorValue*). Esta reutilización reduce el esfuerzo de desarrollo y la redundancia de componentes entre diferentes aplicaciones. Por este motivo, la clase *Sensores Wi-Fi Alrededor* se ha descartado, ya que el *Enlace Wi-Fi* (*WifiLink*) es un sensor más en la relación existente.

11.1.4. Implementación

Los *observadores de contexto* identificados para esta aplicación son los siguientes, implementando cada una de las reglas descritas en la sección de *reglas*:

- **GuidingApplication:** este componente se inicia manualmente desde una interfaz por el usuario de la aplicación (normalmente el dueño del "smartphone", que es el *usuario local*), indicando el profesor con el que se quiere hacer una tutoría. Una vez iniciado, pide al *contenedor de contexto* la *Localización* tanto del usuario como del profesor indicado. Su comportamiento consiste en comprobar si las *Localizaciones* coinciden, y si no es así, proporcionar al usuario la información de la localización del profesor.

- **SensorLocationUpdater:** este componente se inicia como respuesta a la primera petición de *Localización* del usuario del entorno móvil (es decir, el que lleva el "smartphone"). Este componente no es capaz de hallar la *Localización* de ningún otro usuario, porque sólo puede acceder a los sensores presentes en el nodo en el que se ejecuta. Sin embargo, las peticiones sobre la localización del usuario pueden venir también de un nodo remoto. Una vez se inicia, solicita al contexto la relación *SensoresAlrededor* para obtener los *EnlacesWifi* accesibles en cada momento. Entonces, obtiene el *TiempoDeRespuesta* para cada uno de estos sensores. Usando estos tiempos, calcula la *Localización* actual del usuario, haciendo uso de una librería externa y utilizando conocimiento aprendido previamente.

- **SurroundingWifiSensorsUpdater:** este componente se inicia como respuesta a la primera petición de la relación *SensoresAlrededor* del usuario local del entorno. En ese momento, activa los controladores locales capaces de detectar los puntos Wi-Fi cercanos. El componente sigue escuchando estos controladores para actualizar en cada momento qué puntos Wi-Fi existen, y agrega o elimina un *EnlaceWifi* cuando un nuevo punto Wi-Fi se hace en un momento dado accesible o inaccesible. Los sensores que no sean de este tipo no son manipulados por este componente.

- **ResponseTimeUpdater:** este componente se inicia como respuesta a la primera petición del *TiempoDeRespuesta* de un determinado *EnlaceWifi*. En ese momento, inicia un hilo que comprueba periódicamente los tiempos de envío de los paquetes a los distintos puntos Wi-Fi, y actualiza el atributo con el tiempo obtenido.

- **TimetableLocationUpdater:** este componente se inicia como respuesta a la primera petición de la *Localización* de un *Profesor*. Sólo se ejecuta en un nodo controlado por la universidad (al contrario que el nodo del "smartphone"). Ello se debe a que necesita obtener el *Horario* del profesor para calcular su localización, y esta información es sólo accesible localmente por motivos de privacidad. El componente lleva a cabo esta tarea comparando el *Horario* con la hora actual.

Figura 11.2: Representación 3D del escenario "Tutorías Flexibles" con UbikSim.

- **TimetableInfoUpdater:** este componente se inicia como respuesta a la primera petición del *Horario* de un cierto *Profesor*. En ese momento, busca en su base de datos local para obtener un objeto que contiene la información de las horas lectivas o de trabajo del profesor especificado, y con esto actualiza el atributo correspondiente.

El *SensorLocationUpdater* y el *TimetableLocationUpdater* constituyen fuentes de información redundantes. Esta circunstancia es manejada automáticamente por la infraestructura, como se especifica en la Sección 7.2.

11.1.5. Despliegue y pruebas

En este caso hay dos tipos de entorno: un entorno de la universidad, y múltiples entornos móviles correspondientes a los teléfonos móviles de los profesores y los alumnos. La Figura 11.2 muestra una captura de pantalla de la simulación del entorno: uno de los pisos de la facultad.

11.2. Escenario "Talking Agents"

Este caso de estudio se basa en el trabajo "*Talking Agents*" (Fernández-de Alba y Pavón, 2010b). Este trabajo describe una instalación artística inter-

activa donde el espectador se mueve a través de distintas habitaciones. En cada una, el sistema determina la posición del espectador para disparar interacciones con esculturas de barro que representan oráculos. Estas estatuas son los *Talking Agents*, y poseen receptores de voz, dispositivos y componentes software para reconocer el habla y gestionar el diálogo. Este caso es uno de los identificados en el Capítulo 8. Este análisis se va a centrar en el control oportunista de las alternativas de localización. Más concretamente, se van a identificar las operaciones que lleva a cabo el sistema en reacción a ciertos eventos.

11.2.1. Definición del sistema

A continuación se describen algunos requisitos a tener en cuenta en el diseño del SSC. Una representación 3D del entorno asociado se muestra en la Figura 11.3.

1. Las localizaciones en las habitaciones se definen como sectores del suelo. Estas divisiones pueden cambiar en distintos despliegues, y de este modo las posibles localizaciones de los espectadores.

2. Los espectadores pueden ser localizados en cada habitación con distintos tipos de sensores. El sistema hace uso de los sensores disponibles para localizarlos. Los sensores desplegados y su tipo pueden cambiar en tiempo de ejecución. Concretamente, este escenario considera dos tipos de sensores: de distancia y de baldosa. Los sensores de distancia hacen uso de triangulación, y los sensores de baldosa detectan presión en una determinada área del suelo.

3. En caso de redundancia o conflicto entre las localizaciones calculadas, el sistema selecciona aquel cálculo con mayor nivel de "confianza" (*confidence level*).

4. Los fallos en los sensores y los despliegues incompletos pueden crear "puntos ciegos". La aplicación hará lo posible para calcular la localización en estos casos.

5. Durante la ejecución pueden agregarse mecanismos alternativos para la localización.

Llevando a cabo el proceso descrito en la sección anterior, se obtiene un conjunto de elementos de contexto y de observadores que se muestran en la Figura A.13. Algunos de los elementos identificados aquí son "heredados", al igual que en el caso de uso anterior. Éstos serían el tipo de entidad *Persona*, *Sensor*, etc. Además de estos, los elementos de contexto son los siguientes:

- La entidad *TileSensor*, que representa a un sensor de baldosa y tiene los siguientes atributos y relaciones:

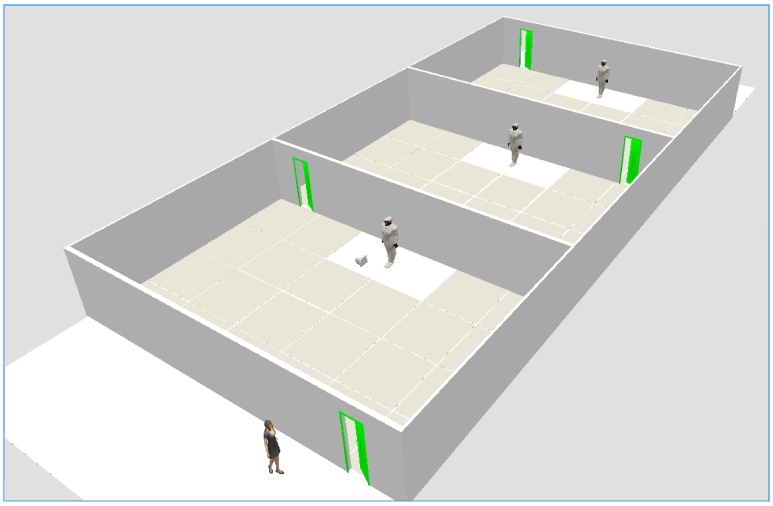

Figura 11.3: Representación 3D del escenario "Talking Agents" con UbikSim.

  - **WeightDetected**: atributo que representa el valor producido por el sensor. Se utiliza para detectar si hay una persona sobre la baldosa.
  - **AdjacentTileSensors**: relación que representa la adyacencia con otros sensores de baldosa. Esta relación puede definirse en el despliegue o tras un periodo de aprendizaje.
- La entidad *MapGraph*, que sintetiza la información de adyacencia de todos los sensores de baldosa y delimita el área de los sectores.
- La entidad *DistanceSensor*, que tiene los siguientes atributos:
  - **DistanceDetected**: atributo que representa el valor producido por el sensor. Es la distancia de un objeto al sensor.
  - **AbsoluteLocation**: atributo que proporciona la localización absoluta y orientación del sensor. Este atributo sirve para determinar el área que está cubriendo el sensor.
- La relación *SectorLocation*, que representa la localización de un único usuario en el entorno en términos de sectores.

Los observadores del contexto identificados en la Figura A.13 son los siguientes:

- **TileSensorLocator**: este observador del contexto proporciona la localización en términos de sector de un usuario, haciendo uso de los sensores de baldosa y la información de adyacencia.

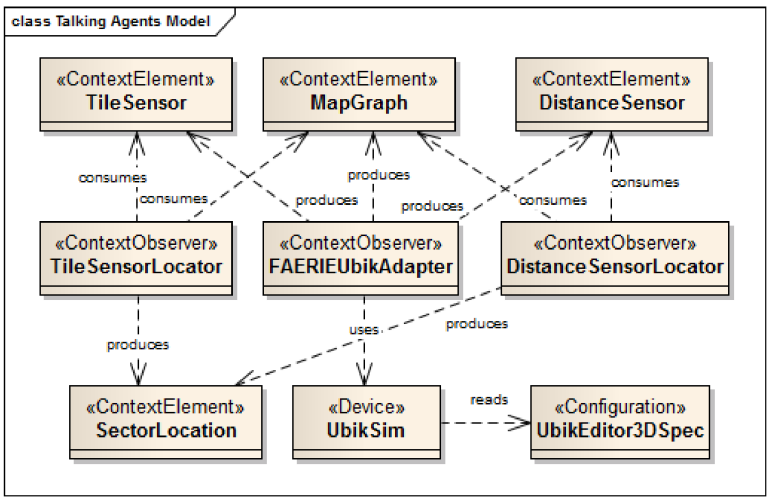

Figura 11.4: Observadores y elementos del contexto para la aplicación “Talking Agents”.

- **FAERIEUbikAdapter**: este observador, ya estudiado en el capítulo anterior, adapta el componente UbikSim. El componente simula el entorno externo a partir de la simulación en UbikSim. Proporciona valores a los sensores de distancia y de baldosa, y proporciona la información sobre la adyacencia.
- **DistanceSensorLocator**: este observador hace uso de la localización absoluta de los distintos sensores y lleva a cabo un proceso de triangulación para hallar el sector en que se encuentra un usuario. Para ello hace uso de la información de adyacencia.

La Figura 11.5 muestra la representación de la entrada y la primera habitación de la instalación. En esta habitación se han desplegado los sensores identificados, pero éstos cubren sólo ciertos puntos. Los círculos blancos representan “puntos ciegos”, es decir, sectores cubiertos por sensores de baldosa rotos. A pesar de que estos no son puntos ciegos cuando la instalación comienza a funcionar, los sensores pueden dejar de funcionar correctamente en ocasiones, o romperse, y sus sectores se convierten en puntos ciegos. Las líneas azules representan cómo está definida la adyacencia entre los distintos sectores.

11.2.2. Ejecución del sistema

La tabla A.1 representa los eventos que se producen en el sistema y cómo los distintos observadores de contexto manipulan el contexto. A continuación, se explican estos eventos de una forma más detallada:

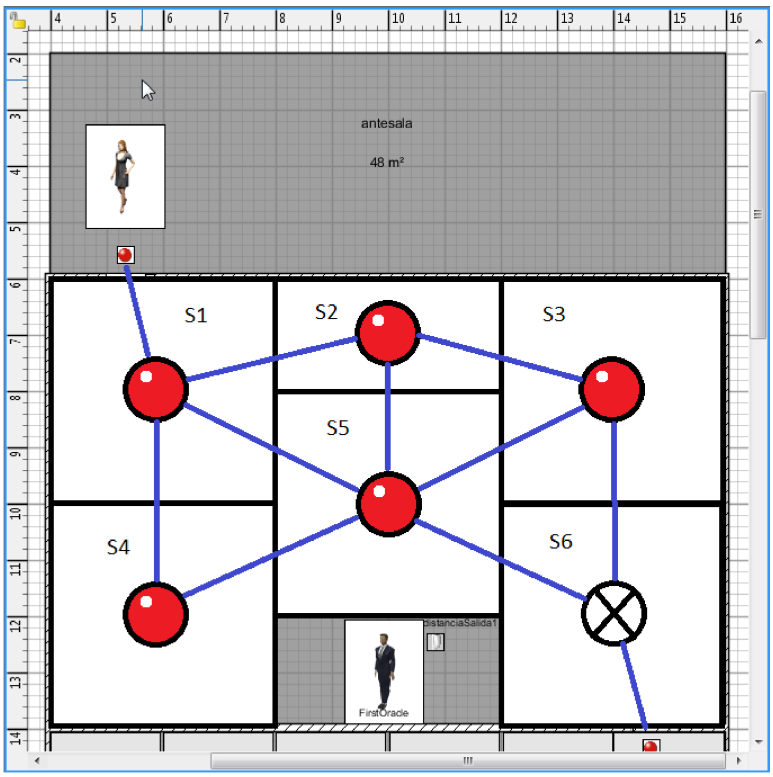

Figura 11.5: Edición en el UbikEditor de la primera habitación del escenario "Talking Agents".

1. Cuando la persona está fuera de la habitación, se considera dentro de un "punto ciego". El *TileSensorLocator* solicita todos los *TileSensor* adyacentes a un punto ciego, de acuerdo a la información en el *MapGraph*. La petición de estos elementos produce la activación de los sensores correspondientes en el entorno (los de baldosa de los sectores 1, 3 y 5). Dado que este componente no puede ofrecer una localización debido a la falta de información, el sistema delega en el siguiente candidato, el *DistanceSensorLocator*. Este componente solicita todos los *DistanceSensor* de la primera habitación. Como no existen sensores de distancia en la entrada, tampoco puede hallar la localización, y ésta se mantiene en un valor indefinido.

2. La persona se mueve al sector 1. En este caso el *TileSensorLocator* ofrece las localizaciones correctas usando la información de los *TileSensor* que cubren estos sectores. Solamente los sensores de baldosa adyacentes están activos. El *observador de alternativa* comprueba que el *TileSensorLocator* que está proporcionando la localización tiene un nivel de confianza mayor que el *DistanceSensorLocator*. Debido a ello,

Evento del entorno	**Dist. Sensor Locator**		**Ubik Adapter**	**Tile Sensor Locator**		**Ubik Adapter**
	precis.	*localiz.*	*Sensores observados*	*precis.*	*localiz.*	*Sensores observados*
está fuera	0.5	*null	distance en habitac. 1	0.6	null	tile en S1, S3, S5
va al sector 1	0.5	null	ninguno	0.6	*sector 1	tile en S1, S2, S4, S5
va al sector 6	0.5	*sector 6	distance en habitac. 1	0.6	null	tile en S1, S3, S5
despliegue en el sec. 6	0.5	null	ninguno	0.6	*sector 6	tile en S3, S5, S6, S7
ruido desciende	0.8	*sector 6	distance en habitac. 1	0.6	null	ninguno

Tabla 11.1: Cambios en el sistema "Talking Agents" como resultado de los eventos del entorno. El asterisco (*) representa qué valor se está seleccionando como válido para actualizar la localización.

desactiva el componente *DistanceSensorLocator*, y los sensores que éste activó. De este modo ya no se observan los sensores de distancia.

3. La persona entra en el sector 6, el cual es un punto ciego. El *TileSensorLocator* solicita de nuevo todos los sensores de baldosa adyacentes a un punto ciego (es decir, los sensores de los sectores 1, 3 y 5), a la espera de que el usuario "reaparezca". Como la localización ha quedado indefinida, el sistema reactiva el componente *DistanceSensorLocator*, reactivando a su vez los sensores de distancia. Como el punto está cubierto por un sensor de distancia, se proporciona la localización correcta.

4. Un nuevo sensor de baldosa se instala en el sector 6. Esta circunstancia es detectada por el sistema. El *TileSensorLocator* reconfigura su funcionamiento con el sector en el *MapGraph*, y vuelve a hallar la localización correcta. Debido a esto, el *DistanceSensorLocator* vuelve a desactivarse de nuevo.

5. Una nueva circunstancia hace que el nivel de confianza del *DistanceSensorLocator* aumente. El *observador de alternativa* es notificado de este evento, y comienza a dar por válida la localización proporcionada por los sensores de distancia frente a los de baldosa. En este proceso, desactiva el *TileSensorLocator*, desactivando a su vez los sensores de baldosa.

11.3. Resultados de implementación

A continuación se analiza el potencial beneficio de hacer uso de FAERIE para el desarrollo de casos de estudio como los descritos anteriormente. El análisis consiste en evaluar cuánto supone el esfuerzo de desarrollo de implementar un sistema sin usar un framework frente a hacerlo con ayuda de FAERIE. Para estimar estos esfuerzos se hará uso del modelo Putnam de estimación de coste, ajustado las variables a las características de este tipo de sistemas. La subsección 11.3.1 describe brevemente el modelo. A continuación, la subsección 11.3.2 explica el análisis.

11.3.1. El modelo Putnam

El modelo Putnam (Putnam y Myers, 1991) es una técnica de estimación de costes para proyectos de software, que ha sido, junto con COCOMO (*COnstructive COst MOdel*) (Boehm et al., 2000), uno de los modelos de estimación paramétrica que más repercusión ha tenido en el mundo de la ingeniería del software. Está basado en la curva de Rayleigh, y describe cómo varía la necesidad de personal al desarrollar proyectos de distinto nivel de complejidad. La ecuación es la siguiente:

$$E = \left[\frac{Tam}{P}\right]^3 \cdot \frac{B}{Tie^4}$$

- E es el esfuerzo total aplicado al proyecto en personas · año.
- *Tam* es el tamaño del producto en miles de LDC (Líneas De Código). Para esta magnitud sólo se cuentan las LDC *efectivas*, es decir, aquellas que quedan en el producto final después de eliminar todo el código que ha sido reutilizado.
- P es la productividad del proceso, es decir, la facilidad para producir determinado tipo de software. Este factor se especifica mediante una tabla que asocia valores a los distintos tipos de sistemas según su dificultad de desarrollo (tabla 4 de Putnam y Myers (1991)).
- *Tie* es el tiempo límite del proyecto en años.
- B es un factor que escala conforme al tamaño del producto (especificado en la tabla 5 de Putnam y Myers (1991)).

La ecuación muestra que, para un tamaño fijo, el esfuerzo es mayor cuanto menor es el tiempo límite del proyecto, y éste va disminuyendo conforme se relaja este tiempo límite. La complejidad de construir un determinado tipo de software, caracterizado por P, también afecta al esfuerzo. Se considera que el desarrollo de sistemas de información con bases de datos, que son

	FAERIE	**Tutorías Flexibles**	***Talking Agents***
proporción de código "útil"	n/a	86 %	74 %
Líneas De Código	7500	1135	620
Esfuerzo (Personas-mes)	5.25 p-m	0.625 p-m	0.312 p-m
Tiempo (meses)	6 meses	1 mes	0.5 meses

Tabla 11.2: Tamaño del código utilizado para implementar los casos de estudio.

los sistemas informáticos de uso más común, es en el que se alcanza mayor productividad. Algunas razones para ello son que existen comunidades de desarrollo amplias, con mucha experiencia, y numerosas herramientas. Por el contrario, es en el desarrollo de software para sistemas empotrados donde la productividad es más baja. El tamaño del software a desarrollar aparece en la ecuación a través de la variable Tam y del factor B. Este último recoge el hecho de que, a medida que crece el tamaño del proyecto, existen saltos no lineales en el esfuerzo requerido para gestionar la complejidad de su tamaño.

11.3.2. Resultados de esfuerzo

Para estudiar el desarrollo con FAERIE siguiendo el modelo de Putnam, partiremos de los datos recogidos en el desarrollo de los casos de estudio descritos previamente en este capítulo. La tabla 11.2 indica el tiempo y esfuerzo de desarrollo y el tamaño de los códigos utilizados para implementar los casos de estudio el propio framework FAERIE. El llamado "código útil" es aquel que especifica lógica de la aplicación, frente al que sirve a propósitos de gestión de distintos aspectos en el uso del framework. Este código podría ser autogenerado en futuras versiones del framework mediante preprocesamiento del código, pudiendo así enfocar el desarrollo en la lógica. Otra manera de ahorrarse la implementación de este código es haciendo uso de un desarrollo dirigido por modelos, tal y como se plantea como trabajo futuro en esta investigación (ver Sección 12.2).

Para evaluar la diferencia en esfuerzo de desarrollo que supone implementar un sistema sin usar un framework para SSC frente a hacerlo con ayuda de FAERIE, se va a estimar la relación de los tamaños de los sistemas. Para ello se van a adoptar las siguientes hipótesis:

- El tamaño del código de FAERIE es adecuado para la funcionalidad que ofrece. Es decir, el diseño del framework y su codificación han sido suficientemente probados y refinados como para constituir una implementación eficiente de la funcionalidad ofrecida, que no incluye prácticamente líneas de código superfluas.
- El tamaño del código de un sistema que no hace uso de FAERIE será, en el caso peor, igual al tamaño del sistema con FAERIE más el del

propio FAERIE (es decir, se implementa un nuevo framework que se utiliza de la misma manera). Sin embargo, esto no será así en el caso medio ya que existen los siguientes casos: que el sistema no requiera toda la funcionalidad que proporciona FAERIE, con lo que sólo se haría una implementación de la parte que proporcionara esa funcionalidad; o que el sistema utilice una arquitectura menos genérica y por tanto éste sea más "compacto", ya que no hará falta tanto código de gestión del sistema. Así, se asume que el tamaño de un sistema sin FAERIE es un porcentaje de la suma del tamaño del sistema con FAERIE más el código del propio framework.

Para el porcentaje anterior se va a utilizar la media de las proporciones de código útil obtenidas en los casos de estudio (aproximadamente un 80 %). Este porcentaje será una función del tamaño del sistema, ya que a mayor tamaño es esperable una mayor reutilización del código destinado a la gestión. Sin embargo, dado que esta función se desconoce y no es estimable por la ausencia de suficientes datos, se va a considerar constante.

- El uso de FAERIE incrementa la productividad del desarrollo respecto al desarrollo sin frameworks de aplicaciones similares. Esto es así porque FAERIE oculta detalles de coordinación y comunicación de los componentes del sistema. De esta forma, el desarrollo de la aplicación se sitúa en un nivel de abstracción más alto que el de aplicaciones similares, reduciendo el esfuerzo para trabajar los detalles de bajo nivel. Además, al ser un código empaquetado se reduce la incidencia de errores de desarrollo en esta parte.

De este modo, el beneficio sobre el esfuerzo conseguido se representa por la siguiente ecuación:

$$\frac{E_{sin}}{E_{con}} = \left[\frac{(Tam_{con} + Tam_{FAERIE}) \cdot 0{,}8 \cdot P_{con}}{Tam_{con} \cdot P_{sin}}\right]^3$$

- E_{sin} es el esfuerzo sin hacer uso de FAERIE y E_{con} es el esfuerzo haciendo uso.

- Tam_{con} es el tamaño de un sistema que hace uso de FAERIE y Tam_{FAERIE} es el tamaño del framework FAERIE.

- P_{sin} es la productividad de desarrollo sin FAERIE y P_{con} es la productividad de desarrollo con FAERIE. Estos valores son constantes, y se han elegido haciendo uso de la tabla 4 de Putnam y Myers (1991). Para P_{sin} se ha escogido un valor de 17.711, que según la tabla equivale al de una aplicación algo más sencilla que un sistema científico, ya que no incluye modelos ni algoritmos excesivamente sofisticados, pero en

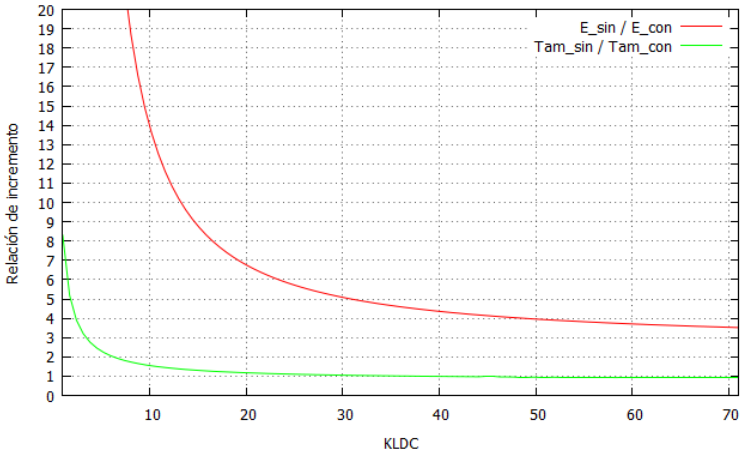

Figura 11.6: Mejora del esfuerzo con respecto al tamaño del software proporcionada por FAERIE.

cambio es más complejo que un sistema de negocio, porque se tienen que manipular diversos componentes distribuidos. Hay que tener en cuenta que este sistema también podría hacer uso de otros middleware distribuidos. Para P_{con} se ha escogido un valor de 28.892, que equivale al de un sistema de negocio. Esto es así porque gracias a FAERIE, se elimina la complejidad de la coordinación de los distintos componentes distribuidos, y los problemas de concurrencia asociados, dejando sólo la gestión de la lógica de la aplicación. De esta forma, la lógica no será mucho más compleja que la de un sistema de información típico, por lo que el valor parece adecuado.

La figura 11.6 muestra el resultado de esta ecuación para distintos tamaños, además del la relación Tam_{sin}/Tam_{con}. Se puede observar que alrededor de 30KLDC, el tamaño del código sin framework se hace más "compacto", hasta llegar a un 80 % del tamaño con framework. Esto es así, porque el código relacionado con las tareas que proporciona FAERIE tiene menos peso en el cómputo general, y gana importancia ese 20 % de ahorro de código de integración. A pesar de ello, se estima que el uso del framework puede reducir el esfuerzo de desarrollo tanto en proyectos pequeños, como en los más grandes, llegando a una reducción a la cuarta parte de esfuerzo. Este incremento se mantiene a pesar del aumento del tamaño del código gracias a que el incremento de productividad supera las ventajas de tener un código más compacto.

11.4. Conclusiones

Este capítulo ha descrito, en primer lugar, cómo se ha desarrollado un sistema real basado en FAERIE. El ejemplo requiere analizar y describir el modelo de contexto, identificar los componentes reutilizables, y desarrollar nuevos *observadores del contexto*. De este modo, se muestra que un prototipo simple se puede especificar fácilmente, haciendo uso de tecnologías conocidas para resolver los problemas que pueden ser complejos.

El proceso utilizado ofrece un buen equilibrio entre simplicidad y generalidad. Ofrece un enfoque de desarrollo más simple que otras alternativas más formales y generales, tales como el trabajo de Sánchez-Pi et al. (2012), y proporciona más guía y correlación con las herramientas de desarrollo proporcionadas.

El segundo caso de estudio se enfoca en mostrar cómo se comporta la arquitectura frente a cambios en la topología como los descritos en la Sección 7.2. Se especifican un conjunto de circunstancias en las que el sistema debe tratar con las fuentes de información redundantes de forma automática. De este modo se ilustra de una manera más concreta la forma de trabajar de FAERIE.

Finalmente, se muestra una estimación del beneficio de utilizar FAERIE para desarrollar este tipo de sistemas. Aunque este beneficio está calculado basándose únicamente en estimaciones y no en valores reales, el resultado ofrece una línea de base para hacer posibles análisis en el futuro. Si se llegan a implementar aplicaciones equivalentes para distintos framework, pueden utilizarse estos cálculos como eje de comparación. De este modo, se puede observar cuánto se desvían los valores reales de los calculados y predecir con mayor exactitud cómo se comportaría cada arquitectura para los distintos casos.

Capítulo 12

Conclusiones

—¿Tan sabio sois?
—Digamos que sí. ¿Habéis oído hablar de Platón? ¿de Aristóteles? ¿de Sócrates?
—Sí.
—Unos incultos.

Westley y Vizzini.
La princesa prometida.

Resumen: En este capítulo se resumen las aportaciones del trabajo y algunas de sus características fundamentales, las líneas de trabajo futuras y las publicaciones que se han realizado al respecto.

12.1. Aportaciones

La aportación principal de esta tesis ha sido una arquitectura de construcción de Sistemas Sensibles al Contexto (SSC) que explora diversas alternativas de enfoque frente a las existentes en la literatura. Esta arquitectura se basa en el modelo de pizarra para llevar a cabo el procesamiento del contexto, aplicado de una forma distribuida. La arquitectura presentada en este trabajo viene acompañada de un marco de desarrollo que facilita la implementación, validación e integración continua de los módulos de construcción.

A pesar de que el modelo de pizarra ya ha sido estudiado en otros trabajos, la forma de abordar la distribución de los componentes requiere en muchos casos elementos centralizados, con los problemas que esto conlleva, o impone restricciones en la definición del procesamiento del contexto asociado. Esta rigidez se traduce en un conjunto de requisitos muy estrictos que debe cumplir una aplicación para poder tratar la información en estas otras

arquitecturas. Además, los trabajos sobre SSC tratan principalmente sobre la obtención y procesamiento del contexto, y no ofrecen mecanismos para orientar el control de las aplicaciones para que estas sean adaptativas. Por último, tampoco se incide en los trabajos existentes en el enfoque que dirige el desarrollo de estas aplicaciones, ni se ofrecen herramientas para facilitarlo.

En este contexto, se ha decidido la creación de una alternativa que explore el modelo de pizarras distribuidas, pero de forma totalmente descentralizada. En la arquitectura presentada, cada unidad de despliegue puede trabajar de forma totalmente autónoma. Además, se permite un rápido prototipado de aplicaciones imponiendo pocas restricciones para utilizar la infraestructura, ya que no se impone una estructura concreta a las aplicaciones en su forma de procesar el contexto. También se ofrecen un conjunto de mecanismos para llevar a cabo adaptación al contexto, mediante la definición de flujos de trabajo o políticas de gestión de cambios de contexto. Concretamente, se ofrece:

- un marco conceptual, que define los términos de Sistema Sensible al Contexto, *entorno*, *contexto*, *contenedor de contexto*, *observador de contexto*, etc.;
- una infraestructura para gestionar el procesamiento del contexto que permite acceder a información de cualquier nivel de abstracción;
- un conjunto de patrones de control para implementar la adaptación de los sistemas a los cambios del contexto;
- y un marco y metodología de desarrollo para facilitar la construcción de SSC.

El **marco conceptual** de la arquitectura establece la definición de un SSC y de sus conceptos asociados. Establece que un SSC se compone de un conjunto de *entornos*, compuestos de una parte *física* y una parte *computacional*. La parte física consta de un *dispositivo principal* y múltiples *dispositivos periféricos*. La parte computacional, que constituye un *nodo* FAERIE, se compone de tres tipos de componentes: *elementos de contexto*, *contenedores de contexto* y *observadores de contexto*. Los elementos de contexto modelan la información con la que trabaja el sistema, y los contenedores y observadores del contexto la manipulan para implementar la funcionalidad de la aplicación. Aparte de esto, se introducen un conjunto de conceptos (algunos ya presentes en la literatura) que sirven para definir diferentes aspectos de un SSC, como el de *enlace activo* y *enlace pasivo*, *observadores anclados*, evaluaciones *top-down* y *bottom-up*, etc.

La **infraestructura** FAERIE ofrece un conjunto de mecanismos para automatizar la gestión del contexto. Los contenedores de contexto se encargan de establecer los canales de comunicación para que los observadores de

contexto manipulen la información. Esto incluye la posible petición de información a entornos remotos y la gestión de su consistencia. Además, FAERIE incluye un modelo y unos observadores de contexto base que son utilizables de forma genérica.

Los **patrones de control** ofrecen un conjunto de mecanismos para implementar comportamientos oportunistas sobre sistemas IAm. Esto implica especificar distintos tipos de comportamientos para distintas condiciones del contexto, de tal manera que la gestión de la evaluación del contexto y del cambio de comportamiento sea automática. En este caso se incluye la definición del comportamiento del sistema por medio de flujos de trabajo. Además, se describe el uso de agentes inteligentes como actores de estos flujos de trabajo sensibles al contexto, de forma que estos tengan acceso a un conjunto de servicios que facilitan su comunicación con el entorno.

Por último, el **marco de desarrollo** incluye un proceso basado en integración continua con plantillas de proyectos y componentes para distintos tipos de despliegues y plataformas. Para este fin se hace uso de la herramienta de construcción de proyectos Apache Maven. Como una plataforma más, el proceso de adaptación incluye la posibilidad de llevar a cabo despliegues en simulación, es decir, con una configuración que hace funcionar la aplicación sobre un entorno 3D utilizando UbikSim. La generación de proyectos se ha estructurado de tal manera que permite pasar automáticamente de un despliegue que haga uso de sensores virtuales (en una simulación) a uno que haga uso de los sensores reales. También incluye el soporte para ejecutar pruebas unitarias y de integración de forma secuencial y automática durante el desarrollo, a fin de facilitar el uso del paradigma de la integración continua.

12.2. Líneas de investigación abiertas y trabajo futuro

El presente trabajo es parte de un esfuerzo más amplio para desarrollar una arquitectura completa que pueda manejar la complejidad de los posibles escenarios en IAm. En este sentido, aún quedan múltiples aspectos por resolver.

En primer lugar, debe definirse una forma correcta de manejar la dimensión temporal de la información. Esto permitiría, por ejemplo, que los sistemas llevaran a cabo razonamiento temporal, es decir, tomar decisiones en base al contexto pasado. FAERIE podría incorporar estos requisitos implementando *observadores de contexto* extra, que se encargarían de registrar los cambios del contexto y proporcionarían acceso a una representación de los mismos.

En segundo lugar, hay que definir claramente el tratamiento de los aspectos de privacidad que hay que tener en cuenta a la hora de manejar los

datos. Esto se haría aprovechando la independencia de los distintos nodos y su *contenedor de contexto* local, y definiendo algún tipo de política que regule la manera de compartir esa información con otros nodos.

En tercer lugar, se ha mencionado que el uso de pizarras distribuidas permite mejorar la robustez y rendimiento de los sistemas. El objetivo es implementar mecanismos para replicar información entre nodos, con el fin de proporcionar métodos de acceso alternativos a cierta información. Otra mejora en el rendimiento sería controlar el uso de la memoria en el contenedor de contexto, limitándola mediante parámetros de configuración. El contenedor de contexto también podría reducir el uso de memoria transfiriendo información que no se utilice o cambie frecuentemente a un dispositivo de almacenamiento secundario.

En cuarto lugar, se contempla hacer uso de herramientas de metamodelado, como INGENME (*INGENIAS Meta-Editor*) (Pavón et al., 2011). Esto permitirá desarrollar un lenguaje gráfico de definición de SSC en base al marco conceptual desarrollado, como se hace por ejemplo en el trabajo de Hassan et al. (2009) para definir sistemas de simulación social. Esto permitiría orientar el desarrollo a uno dirigido por modelos (Schmidt, 2006), dando un mayor nivel de abstracción al proceso.

En quinto lugar, desarrollar en mayor profundidad el uso de agentes inteligentes en los sistemas IAm con el objetivo de llevar a cabo tareas de más alto nivel. Este paso sólo podrá llegar a hacerse una vez que la arquitectura subyacente el nivel de madurez suficiente como para poder construir estructuras cada vez más complejas sobre ella.

Finalmente, ha de tenerse en cuenta, además de la validación cualitativa mediante simulación, alguna estructura de validación cuantitativa, que tenga en cuenta métricas del sistema. El objetivo sería, dados unos requisitos de rendimiento a partir de distintos parámetros, ser capaz de comprobar automáticamente si el sistema los cumple, e incluso que a partir de esta información el sistema sea capaz de autoconfigurarse. Esta validación podría aplicarse también a aspectos de desarrollo, como contabilizar el número de errores producidos por los componentes y el tiempo de desarrollo entre versiones.

12.3. Publicaciones relacionadas

A continuación se presentan las publicaciones surgidas a partir del presente trabajo:

12.3.1. Publicaciones en revistas

- **Caso de uso "Talking Agents"** Fernández-de Alba, J. M. y Pavón, J. Talking Agents: A distributed architecture for interacti-

ve artistic installations. *Integrated Computer-Aided Engineering*, vol. 17(3), páginas 243–259, 2010b. ISSN 1069-2509. (JCR Impact Factor 2010: 2.122)

- **Arquitectura de gestión y procesamiento del contexto** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. Architecture for management and fusion of context information. *Information Fusion*, 2013b. ISSN 1566-2535. (JCR Impact Factor 2012: 2.262). Publicado online, pendiente de publicación en papel

- **Mecanismos de control oportunista** Fernández-de Alba, J. M., Campillo, P., Fuentes-Fernández, R. y Pavón, J. Opportunistic control mechanisms for ambient intelligence worlds. *Expert Systems with Applications*, vol. 41(4), 2014. (JCR Impact Factor 2012: 1.854)

12.3.2. Publicaciones en congresos

- **Arquitectura multimodal con agentes** Fernández-de Alba, J. M. y Pavón, J. Talking agents design on the ICARO framework. En *Intelligent Data Engineering and Automated Learning - IDEAL 2009* (editado por E. Corchado y H. Yin), vol. 5788 de *Lecture Notes in Computer Science*, páginas 494–501. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-04393-2

- **Agregación e interpretación de información** Fernández-de Alba, J. M. y Pavón, J. Recognition and interpretation on Talking Agents. En *Trends in Applied Intelligent Systems* (editado por N. García-Pedrajas, F. Herrera, C. Fyfe, J. M. Benítez y M. Ali), vol. 6096 de *Lecture Notes in Computer Science*, páginas 448–457. Springer Berlin / Heidelberg, 2010a. ISBN 978-3-642-13021-2

- **Arquitectura multimodal orientada a IAm** Fernández-de Alba, J. M. y Pavón, J. Talking Agents in ambient-assisted living. En *Knowledge-Based and Intelligent Information and Engineering Systems* (editado por R. Setchi, I. Jordanov, R. Howlett y L. Jain), vol. 6279 de *Lecture Notes in Computer Science*, páginas 328–336. Springer Berlin / Heidelberg, 2010c. ISBN 978-3-642-15383-9

- **Infraestructura de gestión del contexto** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. A dynamic context-aware architecture for ambient intelligence. En *Advances in Computational Intelligence* (editado por J. Cabestany, I. Rojas y G. Joya), vol. 6692 de *Lecture Notes in Computer Science*, páginas 637–644. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21497-4

- **Patrones de control y adaptación** Fernández-de Alba, J. M., Campillo, P., Fuentes-Fernández, R. y Pavón, J. Opportunistic sensor interpretation in a virtual smart environment. En *Intelligent Data Engineering and Automated Learning - IDEAL 2012* (editado por H. Yin, J. Costa y G. Barreto), vol. 7435 de *Lecture Notes in Computer Science*, páginas 109–116. Springer Berlin / Heidelberg, 2012a. ISBN 978-3-642-32638-7

- **Gestión de flujos de trabajo sensibles al contexto** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. Dynamic workflow management for context-aware systems. En *Ambient Intelligence - Software and Applications* (editado por P. Novais, K. Hallenborg, D. I. Tapia y J. M. C. Rodríguez), vol. 153 de *Advances in Intelligent and Soft Computing*, páginas 181–188. Springer Berlin / Heidelberg, 2012b. ISBN 978-3-642-28782-4

- **Participación de agentes en flujos de trabajo** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. Agent participation in context-aware workflows. En *Hybrid Artificial Intelligent Systems* (editado por J.-S. Pan, M. M. Polycarpou, M. Woźniak, A. C. Carvalho, H. Quintián y E. Corchado), vol. 8073 de *Lecture Notes in Computer Science*, páginas 31–40. Springer Berlin / Heidelberg, 2013a. ISBN 978-3-642-40845-8

Apéndice A

English Summary

A.1. Introduction

Ambient Intelligence (AmI) worlds are composed of multiple devices that are interconnected and tightly integrated with the environment where they are embedded (Remagnino et al., 2005). They provide services to their users based on the information they gather, what allows them minimizing the need of explicit actions by users. This information is called *context* and comes from different sources, e.g., sensors, users' input, historical data, or external sources (Abowd et al., 1999).

Building applications with this functionality implies dealing with the heterogeneity of devices and the information they manage. These applications have also to consider the evolution of user related features, such as location, time, or activities. There are also dynamic changes in system configurations. Device connections can change or fail, modifying the topology and configuration of the system. These changes, in turn, modify the available information, providing new or redundant data, or making impossible to get an accurate representation of the current context. There are also requirements of quality of service, such as performance, cost, or energy consumption, which are important for real-time applications, to improve device autonomy, or to build affordable systems (Baldauf et al., 2007).

AmI (*Ambient Intelligence*, Inteligencia Ambiental) systems must deal with the previous issues requiring as little as possible user intervention for their configuration and functioning. Users should not be aware of all the devices in the environment in order to use it (Kieffer et al., 2009). Therefore, the system itself must be able to adapt dynamically to all those changes, building context representations with available information and making decisions based on the reliability of such representations.

Opportunistic control mechanisms can be used to support the design of AmI applications with these features. Following this approach, when the system receives a goal, it does not solve it immediately, but suspends it.

Then, it monitors changes in the context to detect a suitable moment to solve the goal, i.e., when needed resources and information are available, and the context fulfills certain conditions (Patalano y Seifert, 1997). The observation of the context continues during goal resolution, so the system can suspend the process if the goal becomes unreachable. This behavior implies the existence of mechanisms to facilitate context-awareness: components are able to determine when conditions depending on context change.

Some recent works (Huang et al., 2008) have proposed platforms that support these opportunistic features. They are mostly focused on the layers at the lowest levels of abstraction. These layers are related with establishing opportunistic connections among the nodes in an unstable network (Arnaboldi et al., 2011; Boldrini et al., 2010), and the opportunistic use of unknown remote abstract resources (Conti et al., 2010; Kurz y Ferscha, 2010). A further step is to prepare systems for identifying abstract conditions on their current context in order to produce abstract interpretations and to execute opportunistic behaviors (Challa et al., 2005). This opportunistic information fusion is relevant to AmI applications where there exist behaviors dependent on multiple inputs. Although there are architectural solutions for all these levels individually, there is a lack of architectures including opportunism at every level.

The development of AmI systems does not only require infrastructure for applications, e.g., platforms or libraries, but also tools for that development. One of the most problematic issues is how to test and validate real applications due to their deployment costs. A solution to this problem is simulating most of the involved physical devices and their deployment with virtual spaces. There exist works that develop test simulations using virtual sensors (Park et al., 2007), and others that apply 3D scenarios to manage (Shirehjini, 2005) or demonstrate (Bylund y Espinoza, 2002) smart spaces in specific settings. However, little work has been done regarding general platforms that use 3D scenarios for assisting in the design and validation of AmI systems.

This thesis presents an infrastructure that addresses the previous issues. It includes two elements. There is a library providing the mechanisms for the opportunistic management of context in AmI applications called FAERIE (Framework for AmI: Extensible Resources for Intelligent Environments) (Fernández-de Alba et al., 2012b). This is complemented with UbikSim (Campillo-Sánchez y Botía, 2012), a platform for the simulation of 3D AmI worlds, which supports the testing of FAERIE-based applications. These elements make possible a development process based on the identification of components, their development based on library elements, and testing using initially simulations. This process is the second contribution of this thesis.

In this context, FAERIE provides different services for building AmI worlds (Fernández-de Alba et al., 2012b). AmI components are defined as

context observers that observe and update shared *context containers*, which manage parts of the abstract representation of the real context. These context containers transparently coordinate among them to offer a virtual globally shared representation of the context, which is distributed among different nodes. When a context observer modifies a piece of the context representation, every other context observer interested in that piece of information is made aware of the change, which triggers successive behaviors. FAERIE uses these components to support the development of workflow-based context-aware applications. This implies that applications are designed around the definition of sets of interconnected activities involving different actors (Ardissono et al., 2007).

To illustrate this summary, a case study of an application that belongs to an artistic installation is used. The application guides spectators through different rooms, using sensors to find their positions and speakers to give them instructions.

A.2. Objectives

This section details the objectives of this dissertation. Firstly, it considers those related to the technological and research points of view that help to cope with the already mentioned challenges in the field. Secondly, it considers objectives linked to the development process required to integrate the different technological aspects. This process must guide and assist engineers when producing applications of this kind.

A.2.1. From a technological point of view

The research is oriented towards AmI system architectures and infrastructures for their development. The objective is to develop the FAERIE architecture, as a framework to facilitate the creation of this kind of applications. The concrete objectives for this purpose are the following:

- Create a conceptual framework for AmI systems, i.e., a model of the identifiable relevant elements in a system and how they will behave and relate among them.

- Establish a development infrastructure under the paradigm of "context awareness". The point is to implement its main common mechanisms. This will allow reducing the development effort for these applications, and facilitate being able to focus on domain issues instead of on low-level details. Distributed and dynamic deployments have to be considered in context management. Thus, the architecture and framework must be able to maintain a consistency on the context management

under changes in topology in runtime. In addition, they have also to be able to work in a distributed way.

- Provide patterns and mechanisms to implement functionality for automatic adaptation to context. For those cases in which it is not possible to make adaptation automatically, give clear patterns to implement it using the infrastructure.

A.2.2. From a development framework point of view

The resulting infrastructure has to be delivered and used in a way that facilitates the development and validation of AmI systems. For that reason, the following objectives are considered:

- Make use of development with *Continuous Integration*. This paradigm goes after improving the development cycle. It proposes using tools that facilitate making changes incrementally in code, and evaluating the quality of each part of the software. Technologies used with this paradigm are, for example, unitary testing, module management software, version control, and integration servers. These tools also support creating deployable elements automatically for different target configurations, which is useful in a context where multiple distributed devices may exist.

- Give support to supervised validation of requirements using simulation. As part of the development cycle, developers will have the possibility of testing their applications in a simulated 3D environment. This objective is especially interesting considering the huge cost of implementing real deployments.

- Establish architecture usage guides. They indicate how to make a correct use of the concepts and mechanisms of the infrastructure to obtain the desired results. They adopt the form of lists of questions and tasks necessary to use properly the different libraries and tools. The result is a set of tutorials (Fernández-de Alba, 2013) and manuals on the infrastructure (see Chapter 9 and Section A.5.2). As these guides are not complete development procedures, they are intended for their integration in existing methodologies.

A.3. Methodology

The research process started with a phase of literature study, in which existing works in the field were analyzed. This study allowed extracting and

characterizing the main features considered for AmI applications. These features were used to define the case studies used in this thesis work. The literature study also identified the key differences among existing architectures, and their advantages and disadvantages.

The hypothesis assumed in this work is the following: the problems the target architecture must address can be identified in an incremental way. This was done by defining case studies of growing complexity, by means of aggregating new characteristics to the simpler ones. These cases also served to validate the proposed solutions. Thus, the second phase consisted in the development of the proposed architecture using the case studies to analyze the different alternatives and see which ones offer the best tradeoffs to be integrated int the proposed solution. The following iterative and incremental process was followed:

1. Define a case study according to the typical characteristics of those in the literature, using a minimal set of characteristics.
2. Extend/modify the conceptual framework of the architecture to model the context and the elements of the considered case study.
3. Add/modify the necessary capabilities in the infrastructure to automatize the management of the behaviors present in the case study. Consider the existing alternatives in literature that bring desirable features to the architecture.
4. Reimplement the applications making use of the new capabilities.
5. Extract plausible repeatable patterns of the application and abstract them as new mechanisms.
6. Test the application again and validate.
7. If the result is satisfactory, add new characteristics to the case study.
8. Jump to point 2.

A.4. The FAERIE Framework

A.4.1. Context Management

FAERIE conceives a *context-aware system* S as a set of interconnected *environments* $\varepsilon_1, \ldots, \varepsilon_n$. Each environment ε_i comprehends:

- a *physical space* p_i, which contains a single *main device* d_{i0} and a set of *peripheral devices* $d_{i1}, \ldots, d_{im}$ physically connected to d_{i0}. These devices include the sensors and actuators deployed in the space, as

well as other elements that may provide additional services (e.g., a database connection). The type P is defined as the set of all possible combinations of states of devices in a physical space. The *location and coverage* of p_i are determined by the location, type, and range of the devices with sensing or acting capabilities in it, and it may change over time. This way, many physical spaces from different environments may overlap.

- a *computational space* c_i (also called *node* in a computer network), which is represented by the software framework running on d_{i0}. It contains the components implementing the logics of AmI applications. Concretely, it contains one *context container* κ_i holding the *context model*, and a set of *context observers* $o_{i1}, \ldots, o_{ip}$ that manipulate that model. The type K is defined as the set of all possible combinations of states of the context elements in a context container.

- a set of *known environments* $\varepsilon_{i1}, \ldots, \varepsilon_{iq}$ such that each node $c_{i1}, \ldots, c_{iq}$ share a network with c_i.

Figure A.1 represents this structure. An example of connections among nodes is modeled in Figure A.2. It shows three main devices: a smartphone, a desktop computer, and a laptop. There are several peripheral devices (i.e., sensors and actuators), applications and files. Each main device runs its node, which contains its respective context container and context observers. Some of the context observers drive the peripheral devices.

A context container acts as a blackboard, and the context observers examine or update its information according to certain logic. When there is a change in the representation of the context, a notification is sent to the interested (i.e., subscribed) context observers, which are able to modify their behavior accordingly. This, in turn, may modify the current context or the behavior of the driven devices. Thus, the context observers are responsible of making the state of the context progress over time.

In a formal way, a general context observer o_j acts as a function of the form $o_j : K \times P \to K \times P$. This means that given a state of the physical environment and the context container, the observer generates a new state of the context container and changes the physical environment. Assuming a discrete division of time, Figure A.3 represents this process: $p(t)$ and $k(t)$ are the physical environment and the context state respectively at time t; $o1, \ldots, o5$ are the context observers and $c1, \ldots, c6$ are the context elements. The state of the context container at a given time is the result of the combination of the processes of the observers over the state of the context container and the environment in the previous quantum of time.

Information fusion processes are one of the possible functions that context observers may implement. Concretely, their objective is to combine multiple

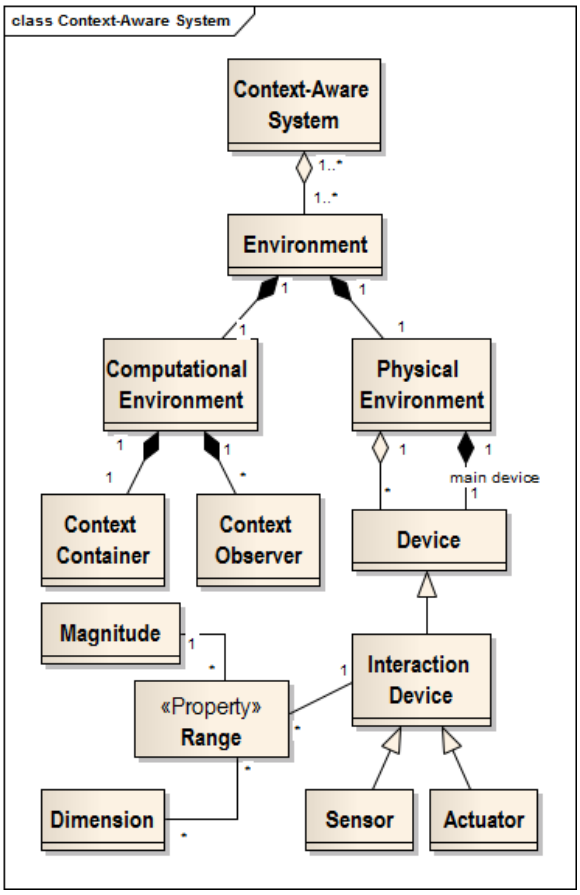

Figura A.1: Schema of a context-aware system in FAERIE.

sources of information in order to: provide a more synthesized and accurate version of the same information (*aggregation*), infer new information making use of heuristics or previous knowledge (*interpretation*), or disambiguate different versions of the same information from multiple sources (Haghighat et al., 2011).

Next subsections describe the mechanisms implemented in FAERIE to support the development of systems that work in the described way. Components are organized following a distributed data-centric blackboard model. This architecture is defined through the interfaces of *context container* and *context observer* components in Section A.4.1.1. The context containers hold information that governs the behavior and interactions among the other components. Section A.4.1.2 describes the structure of this context information. Context observers are able to register themselves with the context container of the node where they run, and to make requests on specific pieces of context. If a request cannot be solved locally within a node, it may be fetched on

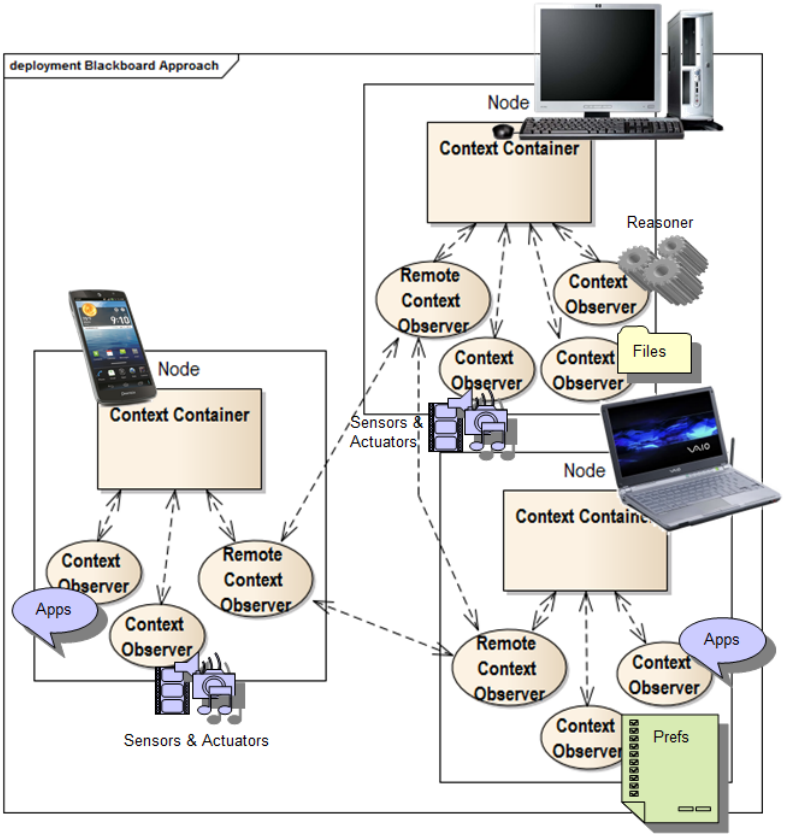

Figura A.2: Example of FAERIE system.

remote known nodes. Section A.4.1.3 explains how this association between consumers and providers takes place at runtime. Finally, Sections A.4.1.4 and A.4.1.5 describe the framework support to implement the actual context processing. The former section is focused on sensing and acting mechanisms (i.e., direct interaction with sensors and actuators respectively), and the later on those to aggregate and propagate context information. This last section also explains the different types of process for context-information fusion that the framework facilitates.

A.4.1.1. Architecture

The software components running in a FAERIE *node* have to implement certain interfaces in order to collaborate. Figure A.4 shows them and their dependencies.

The *context container* component provides methods to access to the object-oriented representation of the current context (see Section A.4.1.2). It

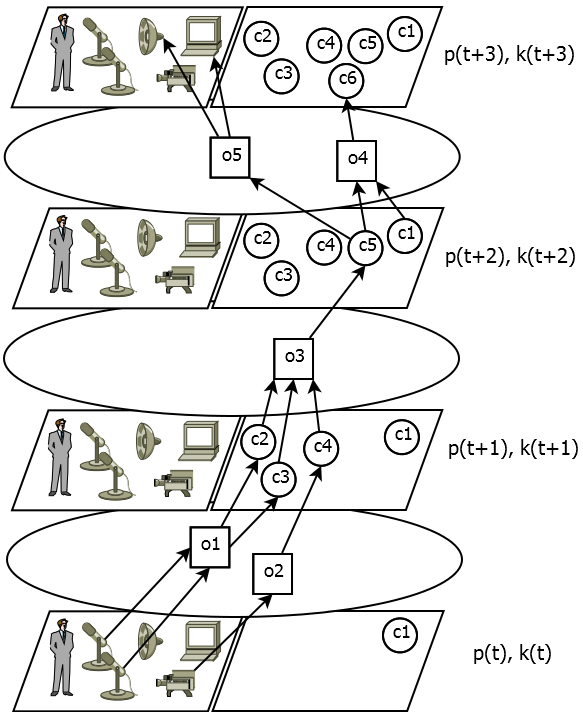

Figura A.3: Process of context change over time.

also associates each element of that representation with the *context observer* components that work on them (see Section A.4.1.3). The *context container* offers these services through the *ContextContainer* interface, and requires the *ContextObserver* interface to inform other components of context changes.

The *context observer* components request and react to changes in the representation of the current context. They can be classified as *grounded* and *abstraction context observers*. *Grounded context observers* access to the external environment using *peripheral devices*, either to modify it according to the context or to update the context according to it. These peripheral devices are sensors and actuators. Observers driving sensors can be modeled as functions of the form $o_{sensor} : K \times P \rightarrow K$, and for actuators as $o_{actuator} : K \rightarrow K \times P$ (see Section A.4.1.4). *Abstraction context observers* do not access the environment, as they only update the context representation following their internal logic and using other information present in the context. That is, abstraction context observers are of the form $o_{abs} : K \rightarrow K$ (see Section A.4.1.5). Both types offer the *ContextObserver* interface and use the *ContextContainer* interface, which link them to *context containers*. They are internally divided into two subcomponents: the *context manager* and the *context-aware component*.

The *context manager* subcomponent implements the *ContextManager* in-

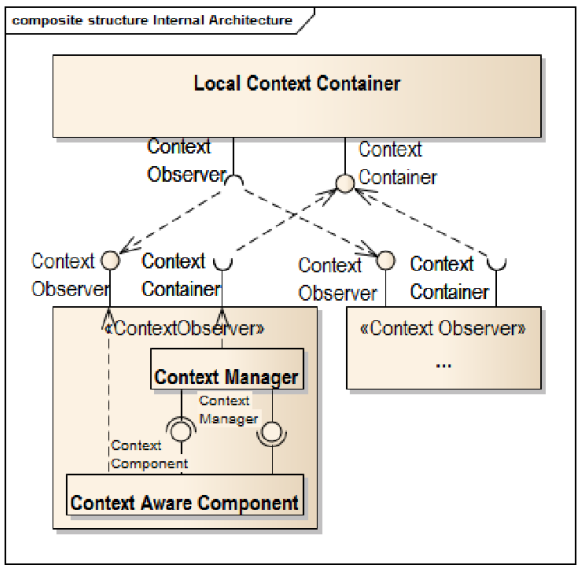

Figura A.4: Dependencies between component interfaces in FAERIE.

terface to fulfill several responsibilities. First, it adapts the *ContextContainer* interface to offer the same functionality but in a simpler way. This relieves the *context-aware component* from some management issues, such as identifying itself in each information request, so it works as if it was the only component accessing to the context information. Second, it obtains and keeps track of the reference to the *context container* of its *node*. Since the *context container* is a dynamic component, it may be undeployed or changed during runtime. The *context manager* hides this to the *context-aware component*. Third, it contains the necessary code to launch the *context observer* in the *node*. This process provides the reference of the *context observer* component to the *context container*.

```
public <T extends ContextElement> T request(T element);

public boolean release(URI elementId);

public boolean subscribe(ContextObserver observer,
    ContextElement contextElement);

public boolean unsubscribe(ContextObserver observer,
    ContextElement contextElement);
```

Listado A.1: Context access methods of the *ContextManager* interface.

Listing A.1 shows the methods of the *ContextManager* interface that

allow the *context-aware component* to manipulate context information. The *request* method retrieves or publishes a certain *context element* in the *context container*. Given an instance of *ContextElement* as the method parameter, if the *context container* contains another instance with the same properties (e.g., class and id), that one is returned; otherwise, the provided instance is published on the *context container* and returned. Publishing means notifying each *context observer* able to *handle* the instance about this addition, in order to allow them to link to it as consumers or providers. Components use the *release* method to declare that they do not need an information element any more. If there is no other component interested in the same context element, the container unpublishes it. Unpublishing means notifying each *context observer* bound to the *ContextElement* about this removal. In this way, these observers can release resources that are no longer needed, for instance, stop monitoring a sensor. Context observers use the *subscribe* and *unsubscribe* methods to declare respectively that they are interested in updates of a certain *ContextElement* or that they are not any more.

The *context-aware* subcomponent implements two interfaces: *ContextObserver* to react to changes in the context; and *ContextComponent* to track its own lifecycle.

```
public int handles(ContextEvent event);

public void elementAdded(ContextEvent event);

public void elementRemoved(ContextEvent event);

public void elementUpdated(ContextEvent event);
```

Listado A.2: Callback methods for context observing in the *ContextObserver* interface.

Listing A.2 shows the methods of the *ContextObserver* interface. The *handles* method is invoked to determine if the component is able to handle the specified event. A *ContextEvent* may refer to the addition, removal or update of *ContextElement*s. This method can return 0 if it does not handle the event, or `HANDLE_R`, `HANDLE_W`, and `HANDLE_RW`. `HANDLE_R` means that the observer is not modifying the element, but it is interested in their changes; `HANDLE_W` implies that the observer modifies the element, but it does not read its state; `HANDLE_RW` is a combination of the previous two. When this method returns other than zero, the *elementAdded*, *elementRemoved*, and *elementUpdated* methods can be called on the component depending on the events to notify. This way, the *context-aware* components are able to track the changes in the context. These methods are used in different activities of the context-aware system, such as context discovering (see Section A.4.1.3),

sensing and acting (see Section A.4.1.4), and context processing (see Section A.4.1.5).

The *context manager* uses the methods of the *ContextComponent* callback interface (see Listing A.3) to allow the *context-aware component* to react to changes in the lifecycle of the different components of the system. The *setContextManager* method is called when the *context container* becomes accessible, providing a reference to the *context manager* for the *context-aware component*. The *activate* method is called after the previous one to trigger the initialization code; the *deactivate* method is called before the shutdown of either the *context-aware component* or the *context container*, to trigger the disposal code. The *unsetContextManager* method is called right before the *context container* becomes inaccessible.

```
public void setContextManager(ContextManager contextManager);

public void activate();

public void deactivate();

public void unsetContextManager(ContextManager contextManager);
```

Listado A.3: The *ContextComponent* interface contains the callback methods for lifecycle management.

A.4.1.2. Context Modeling

Among the alternatives to implement the representation of the context, FAERIE uses an object-oriented approach (Strang y Linnhoff-Popien, 2004), based on the *observer* pattern. Figure A.5 shows the elements that can be found in the computational space of a FAERIE system. Those relative to the modeling of information are placed on the right half, while those relative to its processing are on the left.

At a given moment, each piece of context is represented as a *ContextElement* of one of three possible types: *Entity*, *Attribute*, and *Relationship*. An *Entity* represents a concept of the system's domain vocabulary (e.g., a person, place, or object), and groups all the concrete *Attribute*s and *Relationship*s related to it. An applied *Attribute* holds a piece of information of type T with certain meaning related to an *Entity* of type E (e.g., the age in years of a person, or the name string of a place). An applied *Relationship* links an *Entity* of type S to others of type T according to certain semantics (e.g., a person with the place where s/he is located). The current state of each *Attribute* and *Relationship* is stored as a *ContextValue*. A *ContextValue* contains, besides data of a certain type T (specified by a concrete *Attribute* or *Relationship*), a reference to the source *ContextObserver* (i.e.,

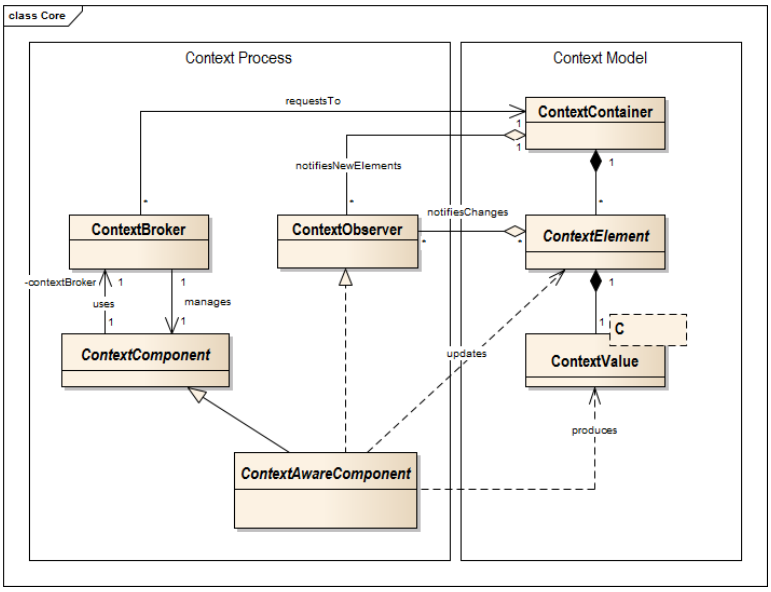

Figura A.5: Basic interfaces and components for context management.

the context observer that produced the value), its creation timestamp (i.e., the moment when the value was obtained), and additional metadata (e.g., a number representing the quality or the cost of the measurement).

In a formal way, the contents of a *context container* κ at a given time t, $\kappa(t) \in K$, $\kappa(t) \subset ContextElement$, are of the form $\kappa(t) = E(t) \cup \mathcal{A}(t) \cup \mathcal{R}(t) \cup O(t)$, where:

- $E(t) \subset Entity$ is the finite set of referenced entities at time t, $e_1, \ldots, e_r$.
- Let $A : Entity \times Attribute$ the set of pairs of entities and attributes. $\mathcal{A}(t) : A \rightarrow ContextValue$ is a function such that $\mathcal{A}(t)(e, \alpha) = v$ being v the value of the attribute α of the entity e at time t.
- Let $R : Entity \times Relationship$ the set of pairs of entities and relationships. $\mathcal{R}(t) : R \rightarrow [ContextValue]$ is a function such that $\mathcal{R}(t)(e, \rho) = [v]$ being $[v]$ the list of entities related with e by the relationship ρ at time t.
- $O(t) \subset ContextObserver$ is the finite set of context observers existing at time t, $o_1, \ldots, o_p$. Each observer o_j has a set of context elements that is *observing* at time t, $Read_j(t) \subseteq \kappa(t)$, and a set of context elements that is *updating* at time t, $Write_j(t) \subseteq \kappa(t)$

The *ContextElement*s provide a method to make an *update attempt* of their values. Listing A.4 shows its pseudocode. Whether the attempt is successful or not depends on the consistency mechanisms implemented by the

context container, as it will be explained in Section A.4.1.5. This way, several context observers may have the same *Attributes* and *Relationships* in their $Write_j(t)$ set at a certain time, but only some of them will actually update their value. Both the attempts and the actual updates will generate events to the subscribed context observers. Using this information, each context observer is able to know what values are generating the others, and use them in their own processing.

```
public synchronized boolean attemptUpdate(
    ContextValue<T> newValue) {
  boolean succeeded = false;
  // Notifies the attempt to the context
  if (this.getContext().publishValue(this, newValue)) {
    // If this attempt is accepted, the value is updated
    this.value = newValue;
    succeeded = true;
  }
  // Notify the observers
  this.notifyAllObservers(UPDATE_ATTEMPT, newValue);
  return succeeded;
}
```

Listado A.4: Method for *Attribute*s.

As *ContextElement*s are classic program objects, handling their information does not require using specific parsers or interpreters. This reduces the processing workload when compared, for instance, with ontology-based models, which are usually represented as XML documents. In addition, this basic model for context can be extended through inheritance, specifying new types of *Entit*ies, *Relationship*s, and *Attribute*s. These new types may include specific constraints on the contained information and even specific methods with their own semantics.

A.4.1.3. Context Discovery

Context discovery is the process that connects *context observer*s that work on the same *context element*. This discovery can happen both when reading or updating information. In the first case, the consumer observer o_i requests some $c_a \in Read_i(t)$, so the system needs to find another observer o_j such that $c_a \in Write_j(t)$ (i.e., provides the required information). In the second case, the updater observer o_i requests some $c_a \in Write_i(t)$, so the system needs to find another observer o_j such that $c_a \in Read_j(t)$ (i.e., that propagates the changes, probably lowering their abstraction level). There are two levels of context discovery: *local*, when it happens within a node, and *remote*, which involves several nodes.

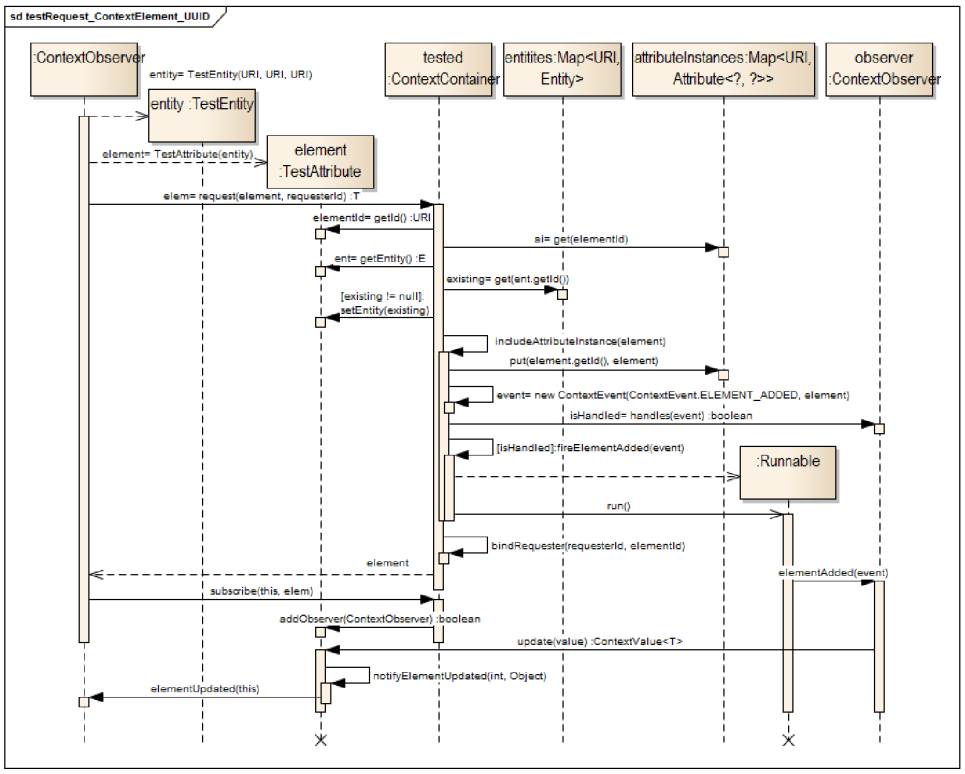

Figura A.6: Broadcasting of the event `request` in a node.

Figure A.6 shows an example of the process for *local context discovery*. A *context observer* that needs a context element makes a *request* to the *context container* using its *ContextContainer* interface. In this example, that request is for the attribute *TestAttribute* of a certain *TestEntity* (an example could be the *ValueDetected* of a certain *Sensor*). When the container receives the request, it has no previous element with the requested features. So, it includes the provided one into the context and tries to locate a suitable *ContextObserver* to provide its information. The container looks for this *ContextObserver* among those registered with it. It invokes the *handles* method (see Section A.4.1.1) of the observers using the new element as argument for the event. The container selects the first observer whose method returns `HANDLE_W` and binds it to the element using its *elementAdded* method (see Section A.4.1.4 for more on this). From that moment on, the bound observer can *attempt update* the *ContextElement* whenever it is necessary. The *ContextElement* will directly notify these updates to all the *ContextObserver*s subscribed to these events (see Section A.4.1.4). In the case that the container is configured to do so, it will forward the request to the *remote context observer* of the *node* if needed. Also, the newly registered *ContextObserver*s will be notified

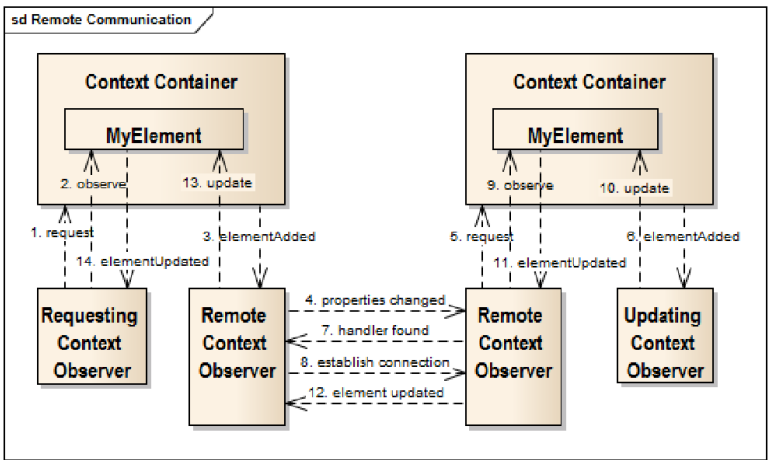

Figura A.7: Broadcasting of the event `request` between nodes.

about the existence of the context element.

A *remote context observer* performs two specific tasks upon its activation. It publishes to the network a reference to itself and detects the references and properties of other *remote context observers* in the network. These properties describe the *ContextElement*s that those *remote context observers* expect others to handle. This exchange of references and properties supports the *remote context discovery* that allows *remote context observer*s to act as proxies for each other.

Figure A.7 depicts the process of *remote context discovery*. It starts when a request has been forwarded to a *remote context observer*. This observer modifies its registered properties to indicate the *ContextElement* that it needs to be handled. Then, it broadcasts these changes to the rest of *remote context observers*. These observers request that *ContextElement* to their own *context containers*. If some of these *nodes* contain a component able to handle that *ContextElement*, their *remote context observers* notify it back to the original requester. This chooses one (or many) of them and establishes a connection between the local and remote copies of the *ContextElement*. From that moment on, every change in the local or remote *ContextElement* will be forwarded to the other through the *remote context observers*.

The use of *remote context observers* makes transparent whether local or remote components update context. The *remote context observer* acts as any other observer in local interactions, while it effectively handles every context element that must be processed remotely by collaboration with other *remote context observers*. This connection creates a "virtual overlapping" among context containers, allowing the system to work as if it was connected to a unique virtual distributed blackboard.

The underlying protocol used to implement remote references and pro-

perty broadcasting in current implementations of FAERIE is Zeroconf (Steinberg y Cheshire, 2005) (a.k.a. *Rendezvous* or *Bonjour*), which uses DNS (Domain Name System) mechanisms. Zeroconf allows publishing and discovering remote objects with associated properties in the local network. It does not require any special configuration of the network parameters. This protocol is implemented transparently by R-OSGi (Rellermeyer et al., 2007) and web services, which have been used as the underlying component platforms in the example implementations of FAERIE. Due to limitations of the Zeroconf protocol, the mechanisms described in this section only work when *nodes* are located within the same local network. When this is not the case, it is necessary to consider a different protocol.

A.4.1.4. Sensing and Acting

Sensing and acting are both extremes of the external interaction of the context-aware system with the environment. Sensing refers to processing new data from sensors and acting to sending new commands to actuators. From the FAERIE perspective, both processes are very similar. They start as part of a context discovery process (see Section A.4.1.3), when the container finds a suitable provider *grounded context observer* for a new *ContextElement*. Then, these processes just differ in which component updates the *ContextElement* and which observes it.

```
public void elementAdded(ContextEvent event) {
  final Attribute attrib = event.getAttributeInstanceArg();
  thread = new Thread(new Runnable() {

    public void run() {
      while (true) {
        // Obtain attribute value from the environment
        // ...
        if (/* value has changed */)
          attrib.update(new DefaultContextValue(value, 1.0));
      }
    }
  });
  thread.start();
}

public void elementRemoved(ContextEvent event) {
  thread.interrupt();
}
```

Listado A.5: Typical code for implementing a *context observer* driving a sensor.

For *sensing*, the selected *context observer* drives a sensor. The container

binds that observer to the *ContextElement* using its *addedElement* method. This call effectively starts the monitoring of the sensor (see Listing A.5). The observer creates a thread where it checks changes in the sensor value, and updates the *ContextElement* related to it when these changes happen. In the same way, the *elementRemoved* callback is used to notify the observer that the *ContextElement* is no longer needed, and hence it stops its monitoring.

In a formal way, the process of sensing acts as a function of the form $o_{sense}(t) : K \times P \to K$ such that $o_{sense}(t)(\{(e, \alpha, \_)\}, p) = \{(e, \alpha, v)\}$ where v is the value provided by the context observer for the attribute α of the entity e under the physical environment state p at time t. The entity e represents a physical device d driven by the observer.

The process of *acting*, as that of sensing, starts with a new *ContextElement*, but here, the process triggers a command for an actuator. When a provider *context observer* publishes some of these *ContextElement*s, the container searches for a suitable *context observer* using the *handles* method. When found, it invokes the *elementAdded* method of the selected *context observer* (see Listing A.6). As this last component drives an actuator, it does not update the *ContextElement*, but *observes* it instead. Then, when the *ContextElement* is updated, the actuator observer receives the *elementUpdated* callback (see Listing A.6), and performs the action corresponding to the value obtained from the element.

```
public void elementAdded(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  contextManager.subscribe(this, attrib);
}

public void elementUpdated(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  ContextValue value = attrib.getValue();
  // Check conditions on the obtained value
  // Perform the required actions
  // ...
}

public void elementRemoved(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  contextManager.unsubscribe(this, attrib);
}
```

Listado A.6: Typical code for implementing a *context observer* driving an actuator.

In a formal way, the process of acting acts as a function of the form $o_{act}(t) : K \to K \times P$ such that $o_{act}(\{(e, \alpha, v)\}) = (\{(e, \alpha, v)\}, p)$, where p is the behavior executed by the context observer in the environment at time t

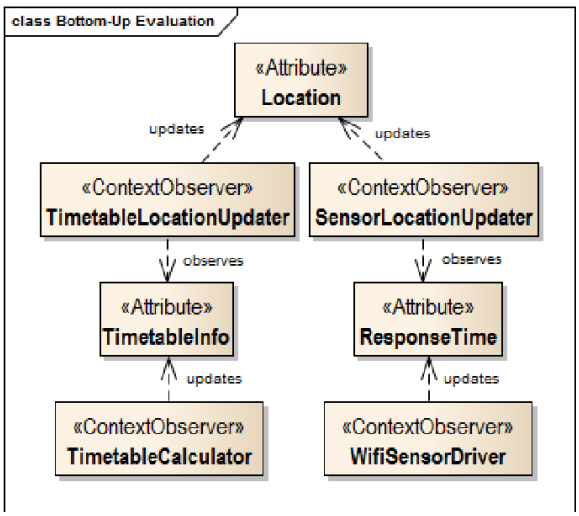

Figura A.8: Runtime relationships between components for the process of a *bottom-up* evaluation.

as a response to the value v of the attribute α in the entity e.

A.4.1.5. Context Processing

Context processing in FAERIE is defined as the transformation of the context model performed by one or more *abstraction context observers*. This transformation may flow in two possible directions: *bottom-up* and *top-down*.

A *bottom-up evaluation* starts when a *context observer* requests one or many *ContextElement*s to *observe* their values, as in a *sensing* process (see Section A.4.1.4). In this case, the provider *context observers* may need subsequent requests to the context, producing a cascade evaluation. The cascade evaluation finishes in *grounded context observers*. From these values, pending calculations are made backwards. These calculations can consist on *aggregation* or *interpretation* of values (Baldauf et al., 2007). Figure A.8 shows an example of the runtime relationships that are established during this process.

Listing A.7 shows an implementation of a context observer involved in a *bottom-up evaluation*. When a certain *Attribute* is published, the method *elementAdded* is called. The calculation of this *Attribute* needs the values of two more *Attribute*s, *s1att* and *s2att*, which in this case come from some sensors with identifiers *s1* and *s2*. Those are requested and then observed by invoking the *request* and *subscribe* methods respectively. When any of these attributes change, the *elementUpdated* method is called. It aggregates their values to calculate the first *Attribute*. When this higher-level *Attribute* is removed from the context, the method *elementRemoved* is called. It releases

```
public void elementAdded(ContextEvent event) {
  updatedAttribute = event.getAttributeInstanceArg();

  // Request the values of lower context elements
  s1 = new MySensor(/* ID of the sensor */);
  s1att = new MySensorAttribute(s1);
  s1att = contextManager.request(s1att);

  s2 = new MyOtherSensor(/* ID of the other sensor */);
  s2att = new MyOtherSensorAttribute(s2);
  s2att = contextManager.request(s2att);

  // ...

  // Observe the values of the lower sensors
  contextManager.subscribe(this, s1att);
  contextManager.subscribe(this, s2att);
  // ...
}

public void elementUpdated(ContextEvent event) {
  Attribute attrib = event.getAttributeInstanceArg();
  if (attrib.equals(s1att))
    value1 = attrib.getValue();
  if (attrib.equals(s2att))
    value2 = attrib.getValue();
  // ...

  // Recalculate context value for the updated attribute,
      depending on the obtained values
  // ...
  updatedAttribute.update(new DefaultContextValue(value, 1.0));
}

public void elementRemoved(ContextEvent event) {
  contextManager.unsubscribe(this, s1att);
  contextManager.release(s1att);
  contextManager.unsubscribe(this, s2att);
  contextManager.release(s2att);
  // ...
}
```

Listado A.7: Typical code to implement a bottom-up evaluation.

the lower-level *Attributes* used before to calculate the other attribute.

In a formal way, the bottom-up evaluation done by an observer is a function of the form $o_{bottomup}(t) : K \rightarrow K$ such that $o_{bottomup}(t)(\{c_0, \ldots, c_n\}) = \{c_0, \ldots, c_n, c_{n+1}\}$ where c_{n+1} is the context element obtained by the context observer using the list of context elements $c_0, \ldots, c_n$ present in the context.

A bottom-up evaluation is an information fusion process if it takes one

of the following concrete forms:

- **Aggregation:** this kind of processing takes multiple pieces of context information and produces a new context element that synthesizes all the information in a more condensed or abstract way. For example, building a global map using the adjacency information of different places belongs to this category. Typical aggregation techniques are machine learning ones, such as naive Bayes classifiers (which summarize large amounts of data as a set of classes determined by the probability of certain features) (West y Harrison, 1997).

- **Interpretation:** this kind of processing is similar to aggregation, but the result is new information that is inferred from the source information, rather than just a synthesized version of it. For example, guessing the location of some object using the values of certain sensors and the information of their deployment. Subsumption inference is a typical interpretation technique (i.e., defining a concept as the conjunction of other simpler concepts) (Kokar et al., 2009).

- **Redundancy/consistency handling:** this kind of processing involves handling different sources for the same information in order to produce a "better" value. The meaning of this "better" depends on certain policies of the system. For instance, if there are two components calculating the location of the same object using different algorithms and resources, this process may take one of the measurements, or a combination of both. Compression techniques are examples of those used to deal with this kind of problem (Taubman y Marcellin, 2001).

The information generated in bottom-up evaluations is consumed by context-aware applications to perform *adaptation* (i.e., modify their external behavior accordingly). An example of this is guiding users in an area depending on their current locations.

The *top-down evaluation* works in the opposite direction. A *context observer* requests *changing* one or more *ContextElement*s. That causes successive requests to *change* other *ContextElement*s. These requests finish in *acting* processes (see Section A.4.1.4). Then, each time the value of the original upper-level *ContextElement* is changed, these changes are notified downwards, ultimately activating the actuators. For this reason, the process is described as *top-down*. This defines a process of propagation of the values of *ContextElement*s towards more concrete values. Figure A.9 shows an example of the runtime relationships that are set up during this process.

Listing A.8 shows an example of implementation of a context observer for a *top-down evaluation*. When certain *Attribute* is published, the method *elementAdded* is called. It adds the *context observer* as an observer of the

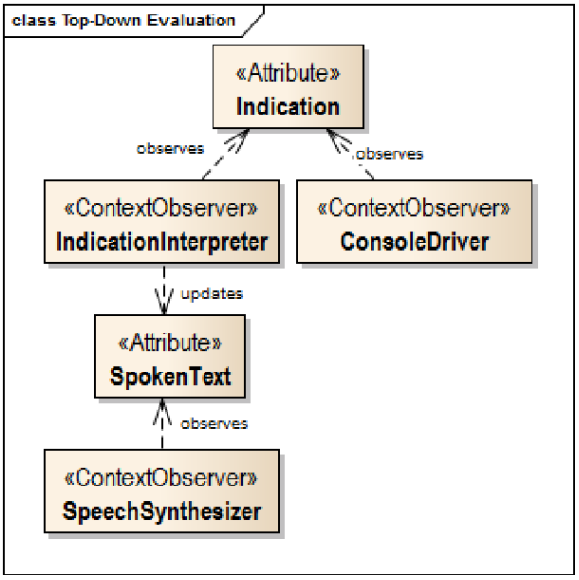

Figura A.9: Runtime relationships between components for the process of a *top-down* evaluation.

parameter higher-level *Attribute* using the *subscribe* method. This *Attribute* provides a value that has to be propagated to two other *Attribute*s that depend on actuators. The *a1att* and *a2att* *Attribute*s are requested, but the triggering *Attribute* is the observed parameter. When this original *Attribute* changes, the *elementUpdated* method is called, and its value is propagated to update the value of the other two *Attribute*s. When the higher-level *Attribute* is removed from the context, the method *elementRemoved* is called. It releases the lower-level *Attributes* that depend on the value of the parameter *Attribute*.

In a formal way, the bottom-up evaluation done by an observer is a function of the form $o_{topdown}(t) : K \rightarrow K$ such that $o_{topdown}(t)(\{c_0\}) = \{c_0, c_1, \ldots, c_n\}$ where $c_1, \ldots, c_n$ are the context elements derived by the observer using the element c_0. As we can see, this process "spreads" the value of a context element to a list of context elements, as it is the inverse process of the bottom-up evaluation.

A.4.2. Opportunistic control

FAERIE adopts an opportunistic control to organize the behavior of its systems. Opportunistic behavior (Patalano y Seifert, 1997) consists in being able to execute tasks when the context meets certain conditions, and doing it in an adapted way to those conditions. This behavior needs control mechanisms to evaluate the conditions, and to suspend and restart tasks depending

```
public void elementAdded(ContextEvent event) {
  observedAttribute = event.getAttributeInstanceArg();

  // Observe the higher context element
  contextManager.subscribe(this, observedAttribute);

  // Request the values of lower context elements
  a1 = new MyActuator(/* ID of the actuator */);
  a1att = new MyActuatorAttribute(a1);
  a1att = contextManager.request(a1att);

  a2 = new MyOtherActuator(/* ID of the other actuator */);
  a2att = new MyOtherActuatorAttribute(a2);
  a2att = contextManager.request(a2att);

  // ...
}

public void elementUpdated(ContextEvent event) {
  observedValue = event.getAttributeInstanceArg().getValue();

  // Recalculate the actuators values depending on the new
      higher value
  // ...
  a1att.update(new DefaultContextValue(value1, 1.0));
  a2att.update(new DefaultContextValue(value2, 1.0));
  // ...
}

public void elementRemoved(ContextEvent event) {
  contextManager.release(a1att);
  contextManager.release(a2att);
  // ...
  contextManager.unsubscribe(this, observedAttribute);
}
```

Listado A.8: Typical code to implement a top-down evaluation

on the changes. This kind of control is relevant to AmI applications because of their dynamic changes and mobile nature. These are the result of the use of technologies as common today as geographic localization or "plug and play" devices.

There are different types of opportunistic control in AmI applications. At the application level, applications perform opportunistic planning. It consists in observing the context in order to exploit the opportunities following the business logic, e.g., notifying the user for certain tasks when walking near the grocery store. At the framework level there are opportunistic computing and opportunistic information fusion.

FAERIE offers these features using the functionality of the components

seen in previous sections of this appendix. The rest of the section provides more details on how it does it: at the framework level (see Section A.4.2.1) and at the application level (see Section A.4.2.2).

A.4.2.1. Framework-level opportunistic behavior

FAERIE does generic opportunistic actions independently of particular applications, e.g., exploit redundant channels of information to be more fault-tolerant. These opportunistic actions depend on policies or preferences defined by system administrators. These preferences establish priorities or filters considering metrics taken from the monitoring of context observers. Examples of these metrics are availability, accuracy, freshness, power consumption, reboot cost, and stability. These generic actions can be considered at two levels: network and computing.

The network level includes the control mechanisms necessary to interconnect distributed components in an unstable network. This implies that the intermediate nodes in the communication are able to store the received messages and send them when the target addressees become reachable. These mechanisms in FAERIE are built using common dynamic connecting technologies such as UPnP (Jeronimo y Weast, 2003) or Zero-Conf (Steinberg y Cheshire, 2005).

A more abstract vision of opportunism appears at the computing level. It implies the use of abstract resources by applications. This makes the framework able to withstand changes in the actual devices connected to the system, as long as they serve to the same purposes. Here, the framework considers event types related with topology changes. They are necessary to reconfigure the information sources and targets in order to be able to provide continuity in the service provision. FAERIE considers the following types of context changes:

- **Addition of context observers to the system.** The possible causes are:
  - *Plug in*: a new context observer has been connected into the system.
  - *Reachable*: a context observer in a remote environment becomes reachable, including its capabilities to calculate new pieces of context information.
  - *Recovery*: a context observer that was not properly working has recovered its normal functioning.
- **Removal of context observers from the system.** The possible reasons are:

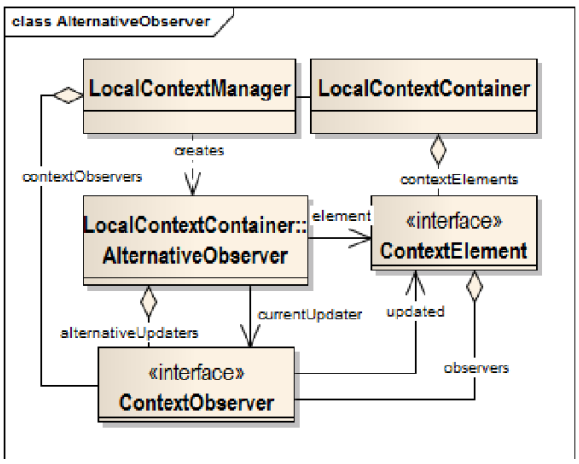

Figura A.10: Reationships of the alternative observer.

- *Unplug*: a context observer is manually disconnected from the system.
- *Unreachable*: a context observer in a remote environment cannot be accessed.
- *Error*: a context observer crashes.

▪ **Change of priority, filtering preferences, or metrics.**

- *Change of metrics*: the metrics monitored from the context observer return values that are incompatible with the system preferences.
- *Change of preferences*: modifications of the system preferences render invalid the updates from certain context observers.

These types of context changes are managed by specific observers called *computational-context observers*. These have certain privileges to adapt the behavior of the system as the environment changes. Their privileges consist in the ability to manipulate directly the information paths, by means of activating or deactivating other context observers. This is necessary to improve the system performance in an opportunistic way by exploiting the different possibilities of the system. Figures A.10 and A.11 respectively illustrate the components and behavior related to one of these observers.

Figure A.10 shows the *alternative observer*, which is a computational-context observer. The context manager creates this alternative observer for each existing context element. The observer performs the activities shown in Figure A.11. It monitors certain metrics in the context in order to select among different observers the one that updates a context element. For this,

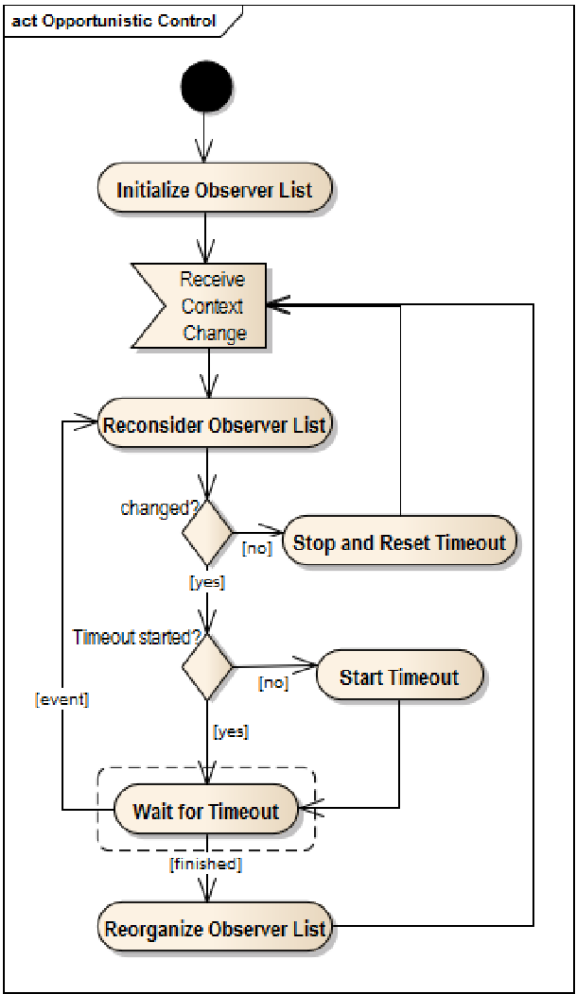

Figura A.11: Activity diagram for the alternative observer.

it maintains a list of candidate context observers to manipulate the context element, and it periodically reconsiders this list.

The list of candidate context observers can adopt different policies:

- **Single candidate:** only one context observer is active after reading the metrics. Typically, it is that preferred according to the observed metrics.
- **N candidates:** a certain number of candidates are active, i.e., producing context values. This alternative produces redundant information, which has to be managed.
- **Everyone is candidate:** using this alternative, there is no need of re-

consideration. Every context observer is held active, and the necessary redundancy management is done. This alternative is expected to produce the worst resource consumption, but may have other advantages, such as lower response times.

Maintaining a list of active candidates for a context element forces the framework to deal with redundant information. There are different potential policies to manage this issue:

- **All updates are valid:** all the updates performed by context observers are accepted, provided that they come temporally ordered.
- **Blacklist/Whitelist:** all the updates performed by observers from the whitelist are accepted, while all performed by members of the blacklist are rejected.
- **Filter by metrics:** only the updates performed by context observers that match the specified metrics preferences are accepted, e.g., filter the values with accuracy higher than a given value.
- **Prioritize by metrics:** if multiple updates are received in the same time slot, those with more preferable values of metrics are chosen (for example, those more accurate).
- **Limit update frequency:** the number of updates by time slot is limited, and all the extra updates are discarded.

In summary, FAERIE provides concrete mechanisms and policies to deal with the identified opportunistic scenarios, which are applied to the lower abstraction layers of the system. These mechanisms manage the information flows of the system, monitoring the possible alternative resources and context changes and readjusting them to the new circumstances, possibly triggering alternative behaviors.

A.4.2.2. Application-level opportunistic behavior

The mechanisms presented in the previous section make possible that FAERIE supports opportunism in the layers at the highest abstraction levels of the architecture. These layers are characterized by components that consider complex context conditions and behavior.

The framework applications are built around the concept of workflow. A workflow defines its actors, resources, and activities, as well as the conditional transitions that connect these activities. The activities can take place in several environments. The use of workflows supports the abstract definition of activities that will be instantiated in different runtime activities depending

on the actual context. For instance, it allows executing tasks when certain resources are available, or when the location changes.

FAERIE applies opportunistic control mechanisms to this kind of workflow. In a running application, context observers continually monitor the alternatives to update the context and choose between them according to events and metrics. When changes occur, their related events are used to trigger the instantiation of workflows or transitions between activities of existing workflows. The triggered activities can imply actions over the environment, and evaluations and processes internal to the system. In this way, the system is able to react in an opportunistic way to the evolution of its environment.

A.5. Experiments

The following subsection describes one of the case studies used to validate the architecture. It introduces the problem (see Section A.5.1), and explains how the framework has been used to implement (see Section A.5.2) and test the related system (see Section A.5.3).

A.5.1. Case study: an artistic installation

The case study is an evolution of the *Talking Agents* artistic installation. This is a convenient case for AmI experimentation because of its flexibility and comprehensibility (Fernández-de Alba y Pavón, 2010b). It considers an intelligent environment where spectators move through different rooms and interact with talking agents. These agents are clay statues with components for speech recognition and dialog management capabilities to communicate with the spectators. In order to guide the spectators' interaction with the installation, the system has to determine the spectators' position, guide their movements to the statues and room exits, and control the interactions with the statues. For the sake of simplicity, this paper only deals with people tracking.

Figure A.12 shows an instance of this installation using the UbikEditor. UbikSim (Campillo-Sánchez y Botía, 2012) is a development kit for the simulation of AmI systems, being UbikEditor its graphical editor for scenarios. UbikSim is the simulation environment adopted in FAERIE for development.

The Talking Agents scenario includes several requirements:

1. Locations in rooms are defined as sectors of their ground. These divisions may change between deployments, and thus the potential spectators' locations.

2. Spectators can be located in each room using different sensors. The system calculates the exact location when required and using the available

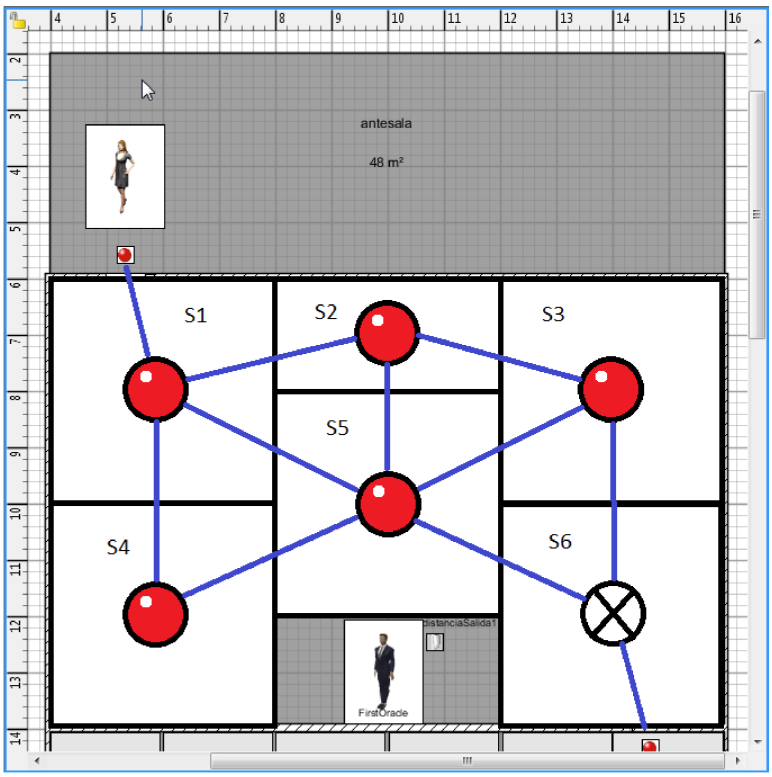

Figura A.12: Map of the first room of the "Talking Agents" scenario in UbikSim.

sensors. The actual sensors used and their kind can therefore change at runtime. Two types of sensors are considered in this scenario: distance sensors and tile sensors. Distance sensors determine location by triangulation using some kind of field or signal, and tile sensors by detecting pressure on a certain area of the floor.

3. In case of redundancy or conflict between the locations calculated using different alternatives, the system selects the one with the highest confidence score.

4. Sensor failures and incomplete deployments may cause blind spots. The application will do its best to determine location under these circumstances.

5. During execution, new sensors or alternative mechanisms to determine location can be added.

A.5.2. Design of the application

FAERIE conceives AmI applications as context-aware workflows. Their design includes three main tasks:

1. Identifying the context elements and modeling the structure of the context representation.
2. Defining the existing activities as workflows, indicating the abstract sub-activities and their relationships with the context information they use and the involved actors.
3. Defining the rules that determine context changes as a function of the changes on other context elements or the physical environment. Rules are implemented as context observer components that work on the representation of the context as context elements.

The following subsections shows how the indicated tasks are used to implement the location and guiding application of the case study.

A.5.2.1. Context identification and modeling

This phase determines several entities, attributes and relationships that are expected to be relevant for the application operation. The elements to take into consideration include:

- Features of physical environments.
- Available components and capabilities.
- Users'characteristics.
- Activities taking place.

These elements make up the context. They are represented in FAERIE as context elements.

An example of these elements in this case study is the information coming from a tile sensor. It is represented by a context element with the same name as the sensor and the following attributes and relationships:

- **WeightDetected**: attribute that holds the value produced by the sensor. This value will be used to detect if there is a person on the tile.
- **AdjacentTileSensors**: relationship that represents adjacency with other TileSensors. This relationship can be defined either at deployment time (as proposed here), or after an automatic learning phase. The complete graph of adjacency is contained in the MapGraph element.

The TileSensorLocator element uses both the attribute and the relationship. They allow it to calculate the location of users in the environment.

A.5.2.2. Workflows definition

The model from the previous phase is used to describe the activities taking place in the AmI system. An activity is specified as a workflow where different actors and entities participate in its subactivities. This process generates new elements to take into consideration for the context model.

The main activity in this case study is interacting through the three rooms of the installation with users for them to experience the intended artistic effects. In each room, there exist in turn other sub-activities that represent the user walking into the room, the system giving indications, and the dialogs between the user and the clay statues. These activities raise new concepts to be considered in the context modeling, such as the user location, the indications given by the system, or the topography of rooms.

A.5.2.3. Context rules definition

The rules represent how the context elements change in response to physical or system changes. They describe how the activities and other entities evolve or are adapted depending on the actual conditions. There are two types of updates: interpretation updates, which reflect actual changes in the real context (e.g., information fusion updates starting in sensors), and decision updates, which reflect the initiative of certain context observers to push the system behavior in certain direction (e.g., updates made by agents to complete running activities).

An example of context rule is how the calculated user location changes as the weight detected by the tile sensors changes. This calculation needs information about the arrangement of the different tile sensors in the room and their data. Therefore, there will exist a context observer dedicated to calculate this location, which will request to the context the values of the different tile sensors, as well as the map graph containing their arrangement.

A.5.2.4. Application structure

Context elements and context observers determine the application structure. For the case study, the diagram in Figure A.13 summarizes it. The types of context observers and the information they use are:

- **TileSensorLocator**: a context observer that provides the user's location. It uses the physical location of the TileSensor, and superimposes the MapGraph to detect position.

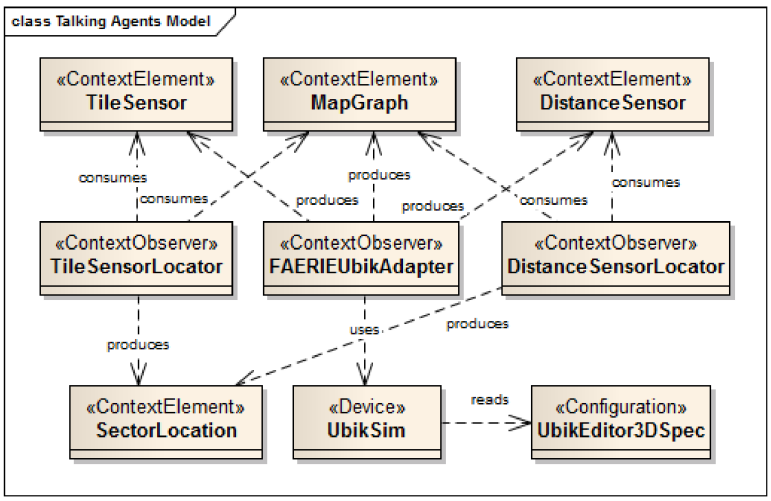

Figura A.13: System structure of the Talking Agents application with FAERIE.

- **FAERIEUbikAdapter**: it adapts the UbikSim component. This component simulates the external environment of a system and provides the MapGraph and DistanceSensor context elements. The MapGraph contains an attribute with a directed graph where each node represents a sector in a room. The DistanceSensor includes two attributes, one with the current state of the sensor and another with its location, orientation, and action range. The data of this last attribute are provided at the moment of its deployment.
- **DistanceSensorLocator**: a context observer that triangulates using the information from DistanceSensors on location, orientation, action range, and current state, and matches the result with the MapGraph to determine the spectator's Location.

These elements allow the system to calculate spectator's locations in an opportunistic way. FAERIE is designed to activate only the context observers that are currently being used. For instance, if there is a context observer already providing the location information, neither the TileSensorLocator nor the DistanceSensorLocator context observers will be activated, and the context elements needed by them will not be requested. In turn, the context observers responsible of calculating those context elements will not be activated either. The mechanism used to choose between observers in this case study is the confidence score. It is a number offered by each context observer to represent the precision of its measurement. As an improvement, the DistanceSensorLocator component only requests the Sensor elements necessary to determine the current location and those in sectors adjacent to it,

where the spectator can move next. If the spectator is in a blind spot, every location next to it is observed.

A.5.3. Testing the application

This section discusses the actual operations performed by the system as reaction to certain events in the first room. Figure A.12 shows its representation. This room has multiple tile sensors that cover the different sectors, as well as several distance sensors, covering only certain spots. White circles represent "blind spots", i.e., sectors covered by broken tile sensors. Though these are not blind spots when the installation starts functioning, sensors may not work properly sometimes or break down, and their sectors become blind spots. The blue lines represent sector adjacency.

Table A.1 represents the events produced in the system, and how the different context observers change the context. The scenario runs as follows:

1. When the person is outside the room, s/he is considered within a virtual "blind spot". The TileSensorLocator requests every TileSensor located next to that blind spot, according to the information in the MapGraph. The request of the context elements produces the activation of the actual sensors in the environment. Since the TileSensorLocator cannot offer a Location due to lack of information, the alternative observer delegates on the DistanceSensorLocator. It requests then the DistanceSensors in the first room. As they do not cover the entrance, the Location cannot be provided, and thus its value is set to null.

2. The person moves into sectors 1, 2, and 3. The TileSensorLocator offers the correct Locations using the information obtained from the TileSensors that cover those sectors. The DistanceSensorLocator is not used, because the SensorLocator is already offering a Location with a higher confidence score. As a consequence, the system deactivates the DistanceSensorLocator, which releases the DistanceSensors.

3. The person enters into sector 6, which is a blind spot. The TileSensorLocator requests again every TileSensor next to that blind spot. As it cannot offer a Location, the alternative observer delegates on the DistanceSensorLocator, which requests again the DistanceSensor. Now, it offers the correct Location.

4. A new tile sensor is deployed in sector 3, which was previously a blind spot. This circumstance is detected thanks to the UPnP (Jeronimo y Weast, 2003) feature of the underlying component framework. The TileSensorLocator reconfigures its map and gets a correct Location for the person. This renders the data from the DistanceSensorLocator unneeded, and thus it is released again, along with the DistanceSensors, as they are not needed at the current person's location.

Tabla A.1: Changes in the system state as a result of environment events. The asterisks represent which value is selected for location.

Environ. event	**Distance Sensor Locator**		**Ubik Adapter**	**Tile Sensor Locator**		**Ubik Adapter**
	confid.	*location*	*Distance Sensors*	*confid.*	*location*	*Tile Sensors*
outside room	0.5	*null	first room	0.6	null	A, B, E
go to sector 1	0.5	null	none	0.6	*sector 1	A, B, D
go to sector 3	0.5	*first room	first room	0.6	null	A, B, E
deploy at sec. 3	0.5	null	none	0.6	*sector 3	B, C, E
room noise decrease	0.8	*sector 3	first room	0.6	null	none

The previous functioning could change if noise in the room decreases making the confidence score of the DistanceSensorLocator higher. Then, the alternative observer would delegate on the DistanceSensorLocator for Location. It would request again the DistanceSensors and get a correct Location. Hence, the TileSensorLocator would be deactivated, which would also release the TileSensors, as another component would be already offering a more reliable result.

A.6. Conclusions

A.6.1. Contributions

The main contribution of this thesis has been an architecture for the development of context-aware systems that explores and integrates different approaches present in literature. This architecture is based on the blackboard model to perform context processing in a distributed way. The architecture presented in this work comes with a development framework, which facilitates the implementation, validation, and continuous integration of the development modules.

When compared with related works, this thesis overcomes several limitations. Despite the blackboard model has already been studied in other works, the way to approach the distribution of components with it requires centralized elements in many cases. This imposes some constraints on the design of context processing that are not inherent to AmI systems, as well as strong requirements on the structure of systems and part of their com-

ponents. Among other issues, these limitations make difficult application interoperability based on context information for systems developed following different architectures. In addition, works about context-aware systems are mainly about obtaining and processing of the context, and do not offer mechanisms to guide the control of the applications to be adaptive. Finally, works in literature do not offer either discussion on the actual development process for these applications, or tools to facilitate it.

To address these issues, this work creates an alternative that explores a distributed blackboard model, but in a completely decentralized way. In the presented architecture, each unit of deployment can work in a fully autonomous way. In addition, it allows a fast prototyping of applications, since it does not impose a concrete structure to process context. This facilitates the incremental building of applications, by adding and refining context observers. A set of mechanisms to perform context adaptation are provided as well. These are based on workflow definitions and policies to manage context changes.

Summarizing, this work offers the following resources to build AmI systems:

- a conceptual framework, which defines the terms *environment*, *context*, *context container*, *context observer*, etc.;
- an infrastructure for the management of context processing that allows accessing its information from any abstraction level;
- a set of control patterns to implement adaptation to context changes;
- and a framework and methodology of development to facilitate building of context-aware systems.

A.6.2. Open research lines and future work

The work presented in this thesis belongs to a reserach line on the development of AmI systems. It tries to provide a complete solution for this purpose. Regarding this, it considers several aspects for future work.

A first issue is the support in context modeling, containers, and observers of additional information and processing requirements. Two aspects considered here are the temporal dimension of information and privacy.

Currently, time can be considered in FAERIE mainly using the timestamp of context elements and the state of workflows. A richer management of this issue should include how to deal with similar context elements over time and their consideration as historical series of information.

Privacy is another key requirement for AmI systems. There are already works that deal with this (Kapadia et al., 2008). The management of privacy has to consider the details regarding the distribution of information.

Opportunistic networks relay on the use of eventual intermediate nodes to transmit information. Control on the choice of these intermediate nodes has to be done to avoid compromising sensitive data. In addition, as opportunistic computing allows executing tasks requested from remote nodes, it is necessary to establish procedures to ensure that the processes do not produce undesirable effects.

A second issue is reducing the need of manual setup for systems. The implementation of automated learning mechanisms would reduce the deployment information to provide on components, therefore simplifying system configuration. These learning processes frequently need a training phase or redundant sources of information, which can be provided using a simulator.

Regarding the development process itself, the two key issues are related to testing and the use of specifications and experience from previous projects adopting model-driven development. Both are intended to reduce the engineers workload.

Even using a simulation environment, engineers still have to validate manually the results of tests. There is need for more automated approaches, as those available in mainstream software development. The use of model checkers and the automatic behaviors present in the UbikSim Kit to validate properties of FAERIE systems are approaches to explore here.

FAERIE research is also exploring the development of AmI systems using model-driven methodologies. There are tools to create domain-specific visual languages and its associated semantic validators and code generators. An example is the INGENME tool (Pavón et al., 2011). This approach would facilitate the integration of systems developed following different approaches in opportunistic context-aware environments. There has been preliminary work in this line using FAERIE and the INGENIAS model (Gómez-Sanz et al., 2008).

A.6.3. Related publications

The main publications produced in the described research are described below.

A.6.3.1. Journal publications

- **Case of study "Talking Agents"** Fernández-de Alba, J. M. y Pavón, J. Talking Agents: A distributed architecture for interactive artistic installations. *Integrated Computer-Aided Engineering*, vol. 17(3), páginas 243–259, 2010b. ISSN 1069-2509. (JCR Impact Factor 2010: 2.122)

- **Architecture for context management and processing** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. Architec-

ture for management and fusion of context information. *Information Fusion*, 2013b. ISSN 1566-2535. (JCR Impact Factor 2012: 2.262). Publicado online, pendiente de publicación en papel

- **Opportunistic control mechanisms** Fernández-de Alba, J. M., Campillo, P., Fuentes-Fernández, R. y Pavón, J. Opportunistic control mechanisms for ambient intelligence worlds. *Expert Systems with Applications*, vol. 41(4), 2014. (JCR Impact Factor 2012: 1.854)

A.6.3.2. Conference publications

- **Multimodal architecture with agents** Fernández-de Alba, J. M. y Pavón, J. Talking agents design on the ICARO framework. En *Intelligent Data Engineering and Automated Learning - IDEAL 2009* (editado por E. Corchado y H. Yin), vol. 5788 de *Lecture Notes in Computer Science*, páginas 494–501. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-04393-2

- **Aggregation and interpretation of information** Fernández-de Alba, J. M. y Pavón, J. Recognition and interpretation on Talking Agents. En *Trends in Applied Intelligent Systems* (editado por N. García-Pedrajas, F. Herrera, C. Fyfe, J. M. Benítez y M. Ali), vol. 6096 de *Lecture Notes in Computer Science*, páginas 448–457. Springer Berlin / Heidelberg, 2010a. ISBN 978-3-642-13021-2

- **Multimodal architecture oriented to AmI** Fernández-de Alba, J. M. y Pavón, J. Talking Agents in ambient-assisted living. En *Knowledge-Based and Intelligent Information and Engineering Systems* (editado por R. Setchi, I. Jordanov, R. Howlett y L. Jain), vol. 6279 de *Lecture Notes in Computer Science*, páginas 328–336. Springer Berlin / Heidelberg, 2010c. ISBN 978-3-642-15383-9

- **Context management infrastructure** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. A dynamic context-aware architecture for ambient intelligence. En *Advances in Computational Intelligence* (editado por J. Cabestany, I. Rojas y G. Joya), vol. 6692 de *Lecture Notes in Computer Science*, páginas 637–644. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21497-4

- **Control and adaptation patterns** Fernández-de Alba, J. M., Campillo, P., Fuentes-Fernández, R. y Pavón, J. Opportunistic sensor interpretation in a virtual smart environment. En *Intelligent Data Engineering and Automated Learning - IDEAL 2012* (editado por H. Yin, J. Costa y G. Barreto), vol. 7435 de *Lecture Notes in Computer Science*, páginas 109–116. Springer Berlin / Heidelberg, 2012a. ISBN 978-3-642-32638-7

- **Context-aware workflow management** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. Dynamic workflow management for context-aware systems. En *Ambient Intelligence - Software and Applications* (editado por P. Novais, K. Hallenborg, D. I. Tapia y J. M. C. Rodríguez), vol. 153 de *Advances in Intelligent and Soft Computing*, páginas 181–188. Springer Berlin / Heidelberg, 2012b. ISBN 978-3-642-28782-4

- **Agent participation in workflows** Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. Agent participation in context-aware workflows. En *Hybrid Artificial Intelligent Systems* (editado por J.-S. Pan, M. M. Polycarpou, M. Woźniak, A. C. Carvalho, H. Quintián y E. Corchado), vol. 8073 de *Lecture Notes in Computer Science*, páginas 31–40. Springer Berlin / Heidelberg, 2013a. ISBN 978-3-642-40845-8

A.7. Main bibliography

- Dey, A. K., Abowd, G. D. y Salber, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, vol. 16(2), páginas 97–166, 2001. ISSN 0737-0024

- Winograd, T. Architectures for context. *Human-Computer Interaction*, vol. 16(2), páginas 401–419, 2001. ISSN 0737-0024

- Chen, H., Finin, T. y Joshi, A. An ontology for context-aware pervasive computing environments. *Knowledge Engineering Review*, vol. 18(3), páginas 197–207, 2003. ISSN 0269-8889

- Hofer, T., Schwinger, W., Pichler, M., Leonhartsberger, G., Altmann, J. y Retschitzegger, W. Context-awareness on mobile devices - the Hydrogen approach. En *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, vol. 9 de *HICSS'03*, página 292. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1874-5

- Korpipaa, P., Mantyjarvi, J., Kela, J., Keranen, H. y Malm, E.-J. Managing context information in mobile devices. *IEEE Pervasive Computing*, vol. 2(3), páginas 42–51, 2003. ISSN 1536-1268

- Ranganathan, A. y Campbell, R. A middleware for context-aware agents in ubiquitous computing environments. En *Middleware 2003* (editado por M. Endler y D. Schmidt), vol. 2672 de *Lecture Notes in Computer Science*, páginas 143–161. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40317-3

- Bardram, J. The Java Context Awareness Framework (JCAF) - a service infrastructure and programming framework for context-aware applications. En *Pervasive Computing* (editado por H. Gellersen, R. Want y A. Schmidt), vol. 3468 de *Lecture Notes in Computer Science*, páginas 98–115. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-26008-0

- Gu, T., Pung, H. K. y Zhang, D. Q. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, vol. 28(1), páginas 1–18, 2005. ISSN 1084-8045

- Haya, P. A., Esquivel, A., Montoro, G., García-herranz, M., Alamán, X., Hervás, R. y Bravo, J. A prototype of context awareness architecture for ambience intelligence at home. En *Proceedings of the 1st International Symposium on Intelligent Environments* (editado por M. Research), vol. 1 de *ISIE'06*, páginas 49–55. Microsoft Research Ltd., Cambridge, MA, USA, 2006

- Henricksen, K., Indulska, J. y MacPhiller, P. Developing context-aware pervasive computing applications: Models and approach. *Pervasive Mobile Computing*, vol. 2(1), páginas 37–64, 2006. ISSN 1574-1192

- Nieto, I., Botía, J. A. y Gómez-Skarmeta, A. F. Information and hybrid architecture model of the OCP contextual information management system. *Journal of Universal Computer Science*, vol. 12(3), páginas 357–366, 2006

- Ejigu, D., Scuturici, M. y Brunie, L. Hybrid approach to collaborative context-aware service platform for pervasive computing. *Journal of Computers*, vol. 3(1), páginas 40–50, 2008

- Buthpitiya, S., Luqman, F., Griss, M., Xing, B. y Dey, A. K. Hermes – a context-aware application development framework and toolkit for the mobile environment. En *Proceedings of the 26th International Conference on Advanced Information Networking and Applications Workshops* (editado por L. Barolli, T. Enokido, F. Xhafa y M. Takizawa), vol. 1 de *WAINA'12*, páginas 663–670. IEEE Computer Society, IEEE Computer Society, Los Alamitos, CA, USA, 2012. ISBN 978-0-7695-4652-0

- Venturini, V., Carbó, J. y Molina, J. M. Methodological design and comparative evaluation of a MAS providing AmI. *Expert Systems with Applications*, vol. 39(12), páginas 10656–10673, 2012. ISSN 0957-4174

Bibliografía

Me convertí en un viajero de la corriente vital y adquirí el conocimiento de los Ancianos.

Sephirot. Final Fantasy VII.

Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M. y Steggles, P. Towards a better understanding of context and context-awareness. En *Handheld and Ubiquitous Computing*, vol. 1707 de *Lecture Notes in Computer Science*, páginas 304–307. Springer Berlin / Heidelberg, London, UK, UK, 1999. ISBN 978-3-540-66550-2.

Aiello, F., Fortino, G., Gravina, R. y Guerrieri, A. A Java-based agent platform for programming wireless sensor networks. *Computer Journal*, vol. 54(3), páginas 439–454, 2011. ISSN 0010-4620.

Fernández-de Alba, J. M. Faerie wiki. `https://www.assembla.com/spaces/a4aal/wiki`, 2013.

Fernández-de Alba, J. M., Campillo, P., Fuentes-Fernández, R. y Pavón, J. Opportunistic sensor interpretation in a virtual smart environment. En *Intelligent Data Engineering and Automated Learning - IDEAL 2012* (editado por H. Yin, J. Costa y G. Barreto), vol. 7435 de *Lecture Notes in Computer Science*, páginas 109–116. Springer Berlin / Heidelberg, 2012a. ISBN 978-3-642-32638-7.

Fernández-de Alba, J. M., Campillo, P., Fuentes-Fernández, R. y Pavón, J. Opportunistic control mechanisms for ambient intelligence worlds. *Expert Systems with Applications*, vol. 41(4), 2014. (JCR Impact Factor 2012: 1.854).

Fernández-de Alba, J. M., Fuentes-Fernández, R. y Pavón, J. A dynamic context-aware architecture for ambient intelligence. En *Advances in Computational Intelligence* (editado por J. Cabestany, I. Rojas y G. Joya), vol. 6692 de *Lecture Notes in Computer Science*, páginas 637–644. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21497-4.

FERNÁNDEZ-DE ALBA, J. M., FUENTES-FERNÁNDEZ, R. y PAVÓN, J. Dynamic workflow management for context-aware systems. En *Ambient Intelligence - Software and Applications* (editado por P. Novais, K. Hallenborg, D. I. Tapia y J. M. C. Rodríguez), vol. 153 de *Advances in Intelligent and Soft Computing*, páginas 181–188. Springer Berlin / Heidelberg, 2012b. ISBN 978-3-642-28782-4.

FERNÁNDEZ-DE ALBA, J. M., FUENTES-FERNÁNDEZ, R. y PAVÓN, J. Agent participation in context-aware workflows. En *Hybrid Artificial Intelligent Systems* (editado por J.-S. Pan, M. M. Polycarpou, M. Woźniak, A. C. Carvalho, H. Quintián y E. Corchado), vol. 8073 de *Lecture Notes in Computer Science*, páginas 31–40. Springer Berlin / Heidelberg, 2013a. ISBN 978-3-642-40845-8.

FERNÁNDEZ-DE ALBA, J. M., FUENTES-FERNÁNDEZ, R. y PAVÓN, J. Architecture for management and fusion of context information. *Information Fusion*, 2013b. ISSN 1566-2535. (JCR Impact Factor 2012: 2.262). Publicado online, pendiente de publicación en papel.

FERNÁNDEZ-DE ALBA, J. M. y PAVÓN, J. Talking agents design on the ICARO framework. En *Intelligent Data Engineering and Automated Learning - IDEAL 2009* (editado por E. Corchado y H. Yin), vol. 5788 de *Lecture Notes in Computer Science*, páginas 494–501. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-04393-2.

FERNÁNDEZ-DE ALBA, J. M. y PAVÓN, J. Recognition and interpretation on Talking Agents. En *Trends in Applied Intelligent Systems* (editado por N. García-Pedrajas, F. Herrera, C. Fyfe, J. M. Benítez y M. Ali), vol. 6096 de *Lecture Notes in Computer Science*, páginas 448–457. Springer Berlin / Heidelberg, 2010a. ISBN 978-3-642-13021-2.

FERNÁNDEZ-DE ALBA, J. M. y PAVÓN, J. Talking Agents: A distributed architecture for interactive artistic installations. *Integrated Computer-Aided Engineering*, vol. 17(3), páginas 243–259, 2010b. ISSN 1069-2509. (JCR Impact Factor 2010: 2.122).

FERNÁNDEZ-DE ALBA, J. M. y PAVÓN, J. Talking Agents in ambient-assisted living. En *Knowledge-Based and Intelligent Information and Engineering Systems* (editado por R. Setchi, I. Jordanov, R. Howlett y L. Jain), vol. 6279 de *Lecture Notes in Computer Science*, páginas 328–336. Springer Berlin / Heidelberg, 2010c. ISBN 978-3-642-15383-9.

ALONSO, G., CASATI, F., KUNO, H. y MACHIRAJU, V. Web services. En *Web Services*, Data-Centric Systems and Applications, páginas 123–149. Springer Berlin Heidelberg, 2004. ISBN 978-3-642-07888-0.

ARDISSONO, L., FURNARI, R., GOY, A., PETRONE, G. y SEGNAN, M. Context-aware workflow management. En *Web Engineering* (editado por L. Baresi, P. Fraternali y G.-J. Houben), vol. 4607 de *Lecture Notes in Computer Science*, páginas 47–52. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-73596-0.

ARMBRUST, M., FOX, A., GRIFFITH, R., JOSEPH, A. D., KATZ, R., KONWINSKI, A., LEE, G., PATTERSON, D., RABKIN, A., STOICA, I. y ZAHARIA, M. A view of cloud computing. *Communications of the ACM*, vol. 53(4), páginas 50–58, 2010. ISSN 0001-0782.

ARNABOLDI, V., CONTI, M. y DELMASTRO, F. Implementation of CAMEO: A context-aware middleware for opportunistic mobile social networks. En *Proceedings of the 12th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, vol. 1 de *WoWMOM'11*, páginas 1–3. IEEE, IEEE Computer Society, Washington, DC, USA, 2011. ISBN 978-1-4577-0352-2.

BALDAUF, M., DUSTDAR, S. y ROSENBERG, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, vol. 2(4), páginas 263–277, 2007. ISSN 1743-8225.

BARDRAM, J. The Java Context Awareness Framework (JCAF) - a service infrastructure and programming framework for context-aware applications. En *Pervasive Computing* (editado por H. Gellersen, R. Want y A. Schmidt), vol. 3468 de *Lecture Notes in Computer Science*, páginas 98–115. Springer Berlin / Heidelberg, 2005. ISBN 978-3-540-26008-0.

BELLIFEMINE, F., POGGI, A. y RIMASSA, G. JADE: a FIPA2000 compliant agent development environment. En *Proceedings of the 5th international conference on Autonomous agents*, vol. 1 de *AGENTS'01*, páginas 216–217. SIGART ACM Special Interest Group on Artificial Intelligence, ACM, New York, NY, USA, 2001. ISBN 1-58113-326-X.

BERGMANN, R. Ambient intelligence for decision making in fire service organizations. En *Ambient Intelligence* (editado por B. Schiele, A. Dey, H. Gellersen, B. Ruyter, M. Tscheligi, R. Wichert, E. Aarts y A. Buchmann), vol. 4794 de *Lecture Notes in Computer Science*, páginas 73–90. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76651-3.

BERGVALL-KAREBORN, B., HOIST, M. y STAHLBROST, A. Concept design with a living lab approach. En *Proceedings of the 42nd Hawaii International Conference on System Sciences*, vol. 1 de *HICSS'09*, páginas 1–10. 2009. ISBN 978-0-7695-3450-3. ISSN 1530-1605.

BOEHM, B. W., CLARK, HOROWITZ, BROWN, REIFER, CHULANI, MADACHY, R. y STEECE, B. *Software Cost Estimation with Cocomo II with*

Cdrom. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edición, 2000. ISBN 0130266922.

BOLDRINI, C., CONTI, M., DELMASTRO, F. y PASSARELLA, A. Context- and social-aware middleware for opportunistic networks. *Journal of Network and Computer Applications*, vol. 33(5), páginas 525–541, 2010. ISSN 1084-8045.

BRATMAN, M. E. *Intention, Plans, and Practical Reason*. Cambridge University Press, 1999. ISBN 1575861925.

BUTHPITIYA, S., LUQMAN, F., GRISS, M., XING, B. y DEY, A. K. Hermes – a context-aware application development framework and toolkit for the mobile environment. En *Proceedings of the 26th International Conference on Advanced Information Networking and Applications Workshops* (editado por L. Barolli, T. Enokido, F. Xhafa y M. Takizawa), vol. 1 de *WAINA'12*, páginas 663–670. IEEE Computer Society, IEEE Computer Society, Los Alamitos, CA, USA, 2012. ISBN 978-0-7695-4652-0.

BYLUND, M. y ESPINOZA, F. Testing and demonstrating context-aware services with Quake III Arena. *Communications of the ACM*, vol. 45(1), páginas 46–48, 2002. ISSN 0001-0782.

CAMPILLO-SÁNCHEZ, P. y BOTÍA, J. Simulation based software development for smart phones. En *Ambient Intelligence - Software and Applications* (editado por P. Novais, K. Hallenborg, D. I. Tapia y J. M. C. Rodríguez), vol. 153 de *Advances in Intelligent and Soft Computing*, páginas 243–250. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-28782-4.

CAMPILLO-SANCHEZ, P., SERRANO, E. y BOTÍA, J. A. Testing context-aware services based on smartphones by agent based social simulation. *Journal of Ambient Intelligence and Smart Environments*, vol. 5, páginas 311–330, 2013.

CAMPUZANO, F., GARCÍA-VALVERDE, T., GARCÍA-SOLA, A. y BOTÍA, J. A. Flexible simulation of ubiquitous computing environments. En *Ambient Intelligence - Software and Applications* (editado por P. Novais, D. Preuveneers y J. Corchado), vol. 92 de *Advances in Intelligent and Soft Computing*, páginas 189–196. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-19936-3.

CHALLA, S., GULREZ, T., CHACZKO, Z. y PARANESHA, T. Opportunistic information fusion: a new paradigm for next generation networked sensing systems. En *Proceedings of the 8th International Conference on Information Fusion*, vol. 1 de *FUSION'05*, páginas 720–727. IEEE Computer Society, 2005. ISBN 0-7803-9286-8.

CHEN, H., FININ, T. y JOSHI, A. An ontology for context-aware pervasive computing environments. *Knowledge Engineering Review*, vol. 18(3), páginas 197–207, 2003. ISSN 0269-8889.

CONTI, M., GIORDANO, S., MAY, M. y PASSARELLA, A. From opportunistic networks to opportunistic computing. *IEEE Communications Magazine*, vol. 48(9), páginas 126–139, 2010. ISSN 0163-6804.

DEY, A. K., ABOWD, G. D. y SALBER, D. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Human-Computer Interaction*, vol. 16(2), páginas 97–166, 2001. ISSN 0737-0024.

EJIGU, D., SCUTURICI, M. y BRUNIE, L. Hybrid approach to collaborative context-aware service platform for pervasive computing. *Journal of Computers*, vol. 3(1), páginas 40–50, 2008.

GAMMA, E., HELM, R., JOHNSON, R. y VLISSIDES, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

GARCÍA-VALVERDE, T., CAMPUZANO, F., SERRANO, E., VILLA, A. y BOTÍA, J. A. Simulation of human behaviours for the validation of ambient intelligence services: A methodological approach. *Journal of Ambient Intelligence and Smart Environments*, vol. 4(3), páginas 163–181, 2012. ISSN 1876-1364.

GASCUEÑA, J., FERNÁNDEZ-CABALLERO, A. y GARIJO, F. Using icaro-t framework for reactive agent-based mobile robots. En *Advances in Practical Applications of Agents and Multiagent Systems* (editado por Y. Demazeau, F. Dignum, J. Corchado y J. Pérez), vol. 70 de *Advances in Intelligent and Soft Computing*, páginas 91–101. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12383-2.

GÓMEZ-SANZ, J. J., FERNÁNDEZ, C. R. y ARROYO, J. Model driven development and simulations with the INGENIAS agent framework. *Simulation Modelling Practice and Theory*, vol. 18(10), páginas 1468–1482, 2010. ISSN 1569-190X.

GÓMEZ-SANZ, J. J., FUENTES, R., PAVÓN, J. y GARCÍA-MAGARIÑO, I. INGENIAS development kit: a visual multi-agent system development environment. En *Proceedings of the 7th International Joint conference on Autonomous Agents and Multiagent Systems*, vol. 1 de *AAMAS'08*, páginas 1675–1676. ACM Association for Computing Machinery and AAAI Association for the Advancement of Artifical Intelligence, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, 2008.

GRIOL, D., CARBÓ, J. y MOLINA, J. M. Optimizing dialog strategies for conversational agents interacting in AmI environments. En *Ambient Intelligence - Software and Applications* (editado por P. Novais, K. Hallenborg, D. I. Tapia y J. M. C. Rodríguez), vol. 153 de *Advances in Intelligent and Soft Computing*, páginas 93–100. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-28782-4.

GU, T., PUNG, H. K. y ZHANG, D. Q. A service-oriented middleware for building context-aware services. *Journal of Network and Computer Applications*, vol. 28(1), páginas 1–18, 2005. ISSN 1084-8045.

HAGHIGHAT, M. B. A., AGHAGOLZADEH, A. y SEYEDARABI, H. Multi-focus image fusion for visual sensor networks in dct domain. *Computers and Electrical Engineering*, vol. 37(5), páginas 789–797, 2011. ISSN 0045-7906.

HAN, J., CHO, Y., KIM, E. y CHOI, J. A ubiquitous workflow service framework. En *Computational Science and Its Applications - ICCSA 2006* (editado por M. Gavrilova, O. Gervasi, V. Kumar, C. Tan, D. Taniar, A. Laganá, Y. Mun y H. Choo), vol. 3983 de *Lecture Notes in Computer Science*, páginas 30–39. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-34077-5.

VAN HARMELEN, F., PATEL-SCHNEIDER, P. F. y HORROCKS, I. Reference description of the DAML+OIL (march 2001) ontology markup language. `http://www.daml.org/2001/03/reference.html`, 2001.

HASSAN, S., FUENTES-FERNÁNDEZ, R., GALÁN, J., LÓPEZ-PAREDES, A. y PAVÓN, J. Reducing the modeling gap: On the use of metamodels in agent-based simulation. En *Proceedings of the 6th Conference of the European Social Simulation Association*, vol. 1 de *ESSA'09*, páginas 1–13. 2009.

HAWKINSON, L. The representation of concepts in OWL. En *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, vol. 1 de *IJCAI'75*, páginas 107–114. The International Joint Conferences on Artificial Intelligence, Inc., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1975.

HAYA, P. A., ESQUIVEL, A., MONTORO, G., GARCÍA-HERRANZ, M., ALAMÁN, X., HERVÁS, R. y BRAVO, J. A prototype of context awareness architecture for ambience intelligence at home. En *Proceedings of the 1st International Symposium on Intelligent Environments* (editado por M. Research), vol. 1 de *ISIE'06*, páginas 49–55. Microsoft Research Ltd., Cambridge, MA, USA, 2006.

HENRICKSEN, K., INDULSKA, J. y MACPHILLER, P. A software engineering framework for context-aware pervasive computing. En *Proceedings*

of the 2nd IEEE International Conference on Pervasive Computing and Communications, vol. 1 de *PERCOM'04*, páginas 77–86. IEEE Technical Committee on Computer Communications (TCCC) and Parallel Processing (TCPP), IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2090-1.

HENRICKSEN, K., INDULSKA, J. y MACPHILLER, P. Developing context-aware pervasive computing applications: Models and approach. *Pervasive Mobile Computing*, vol. 2(1), páginas 37–64, 2006. ISSN 1574-1192.

HOFER, T., SCHWINGER, W., PICHLER, M., LEONHARTSBERGER, G., ALTMANN, J. y RETSCHITZEGGER, W. Context-awareness on mobile devices - the Hydrogen approach. En *Proceedings of the 36th Annual Hawaii International Conference on System Sciences*, vol. 9 de *HICSS'03*, página 292. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1874-5.

HONG, J. I. y LANDAY, J. A. An infrastructure approach to context-aware computing. *Human-Computer Interaction*, vol. 16, páginas 287–303, 2001. ISSN 0737-0024.

HUANG, C.-M., CHAN LAN, K. y TSAI, C.-Z. A survey of opportunistic networks. En *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications - Workshops*, vol. 1 de *AINAW'08*, páginas 1672–1677. IEEE Computer Society, TCDP, IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3096-3.

JERONIMO, M. y WEAST, J. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press, 2003. ISBN 0971786119.

KALYANPUR, A., PASTOR, D. J., BATTLE, S. y PADGET, J. Automatic mapping of owl ontologies into java. En *Proceedings of 16th International Conference on Software Engineering and Knowledge Engineering (SEKE)* (editado por G. R. F. Maurer), vol. 1 de *SEKE'04*, páginas 98–103. 2004.

KAPADIA, A., TRIANDOPOULOS, N., CORNELIUS, C., PEEBLES, D. y KOTZ, D. AnonySense: Opportunistic and privacy-preserving context collection. En *Pervasive Computing* (editado por J. Indulska, D. Patterson, T. Rodden y M. Ott), vol. 5013 de *Lecture Notes in Computer Science*, páginas 280–297. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-79576-6.

KIEFFER, S., LAWSON, J.-Y. y MACQ, B. User-centered design and fast prototyping of an ambient assisted living system for elderly people. En *Proceedings of the 6th International Conference on Information Technology: New Generations*, vol. 1 de *ITNG'09*, páginas 1220–1225. 2009.

KOKAR, M. M., MATHEUS, C. J. y BACLAWSKI, K. Ontology-based situation awareness. *Information Fusion*, vol. 10(1), páginas 83–98, 2009. ISSN 1566-2535.

KORPIPAA, P., MANTYJARVI, J., KELA, J., KERANEN, H. y MALM, E.-J. Managing context information in mobile devices. *IEEE Pervasive Computing*, vol. 2(3), páginas 42–51, 2003. ISSN 1536-1268.

KURZ, M. y FERSCHA, A. Sensor abstractions for opportunistic activity and context recognition systems. En *Smart Sensing and Context* (editado por P. Lukowicz, K. Kunze y G. Kortuem), vol. 6446 de *Lecture Notes in Computer Science*, páginas 135–148. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-16981-6.

LACOUTURE, J., GASCUEÑA, J. M., GLEIZES, M.-P., GLIZE, P., GARIJO, F. J. y FERNÁNDEZ-CABALLERO, A. ROSACE: Agent-based systems for dynamic task allocation in crisis management. En *Advances on Practical Applications of Agents and Multi-Agent Systems* (editado por Y. Demazeau, J. P. Müller, J. M. C. Rodríguez y J. B. Pérez), vol. 155 de *Advances in Intelligent and Soft Computing*, páginas 255–259. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-28785-5.

LUKE, S., CIOFFI-REVILLA, C., PANAIT, L., SULLIVAN, K. y BALAN, G. MASON: A multiagent simulation environment. *Simulation*, vol. 81(7), páginas 517–527, 2005. ISSN 0037-5497.

MASSOL, V. y O'BRIEN, T. *Maven: A Developer's Notebook (Developer's Notebooks)*. O'Reilly Media, Inc., 2005. ISBN 0596007507.

NIETO, I., BOTÍA, J. A. y GÓMEZ-SKARMETA, A. F. Information and hybrid architecture model of the OCP contextual information management system. *Journal of Universal Computer Science*, vol. 12(3), páginas 357–366, 2006.

PARK, J., MOON, M., HWANG, S. y YEOM, K. CASS: A context-aware simulation system for smart home. En *Proceedings of the 5th International Conference on Software Engineering Research, Management Applications*, vol. 1 de *SERA'07*, páginas 461–467. ACIS, IEEE Computer Society, Washington, DC, USA, 2007. ISBN 0-7695-2867-8.

PATALANO, A. L. y SEIFERT, C. M. Opportunistic planning: Being reminded of pending goals. *Cognitive Psychology*, vol. 34(1), páginas 1–36, 1997. ISSN 0010-0285.

PAVÓN, J., GÓMEZ-SANZ, J. y PAREDES, A. The SiCoSSyS approach to SoS engineering. En *Proceedings of the 6th International Conference System of Systems Engineering*, vol. 1 de *SoSE'11* , páginas 179–184. IEEE Computer Society, IEEE Computer Society, 2011. ISBN 978-1-61284-783-2.

PELUSI, L., PASSARELLA, A. y CONTI, M. Opportunistic networking: data forwarding in disconnected mobile ad hoc networks. *IEEE Communications Magazine*, vol. 44(11), páginas 134–141, 2006. ISSN 0163-6804.

PLASIL, F. y STAL, M. An architectural view of distributed objects and components in CORBA, Java RMI and COM/DCOM. *Software-Concepts and Tools*, vol. 19(1), páginas 14–28, 1998.

POPE, A. L. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-63386-8.

PUTNAM, L. H. y MYERS, W. *Measures for Excellence: Reliable Software on Time, Within Budget*. Prentice Hall Professional Technical Reference, 1st edición, 1991. ISBN 0135676940.

RANGANATHAN, A. y CAMPBELL, R. A middleware for context-aware agents in ubiquitous computing environments. En *Middleware 2003* (editado por M. Endler y D. Schmidt), vol. 2672 de *Lecture Notes in Computer Science*, páginas 143–161. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-40317-3.

RANGANATHAN, A. y MCFADDIN, S. Using workflows to coordinate web services in pervasive computing environments. En *Proceedings of the 2nd IEEE International Conference on Web Services*, vol. 1 de *ICWS'04*, páginas 288–295. IEEE Computer Society Technical Community for Services Computing (TCSC), IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2167-3.

RELLERMEYER, J., ALONSO, G. y ROSCOE, T. R-OSGi: Distributed applications through software modularization. En *Middleware 2007* (editado por R. Cerqueira y R. Campbell), vol. 4834 de *Lecture Notes in Computer Science*, páginas 1–20. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76777-0.

REMAGNINO, P., HAGRAS, H., MONEKOSSO, N. y VELASTIN, S. Ambient intelligence. En *Ambient Intelligence* (editado por P. Remagnino, G. Foresti y T. Ellis), páginas 1–14. Springer New York, 2005. ISBN 978-0-387-22991-1.

RODRÍGUEZ, S., PAZ, J., SÁNCHEZ, P. y CORCHADO, J. Context-aware agents for people detection and stereoscopic analysis. En *Trends in Practical Applications of Agents and Multiagent Systems* (editado por Y. Demazeau, F. Dignum, J. Corchado, J. Bajo, R. Corchuelo, E. Corchado, F. Fernández-Riverola, V. Julián, P. Pawlewski y A. Campbell), vol. 71 de *Advances in Intelligent and Soft Computing*, páginas 173–181. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12432-7.

ROGGEN, D., FORSTER, K., CALATRONI, A., HOLLECZEK, T., FANG, Y., TROSTER, G., LUKOWICZ, P., PIRKL, G., BANNACH, D., KUNZE, K., FERSCHA, A., HOLZMANN, C., RIENER, A., CHAVARRIAGA, R. y DEL R. MILLÁN, J. OPPORTUNITY: Towards opportunistic activity and context recognition systems. En *Proceedings of the 10th IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks - Workshops*, vol. 1 de *WoWMoM'09*, páginas 1–6. 2009.

SÁNCHEZ-PI, N., CARBÓ, J. y MOLINA, J. M. A knowledge-based system approach for a context-aware system. *Knowledge-Based Systems*, vol. 27, páginas 1–17, 2012. ISSN 0950-7051.

SÁNCHEZ-PI, N., MANGINA, E., CARBÓ, J. y MOLINA, J. Multi-agent system (mas) applications in ambient intelligence (ami) environments. En *Trends in Practical Applications of Agents and Multiagent Systems* (editado por Y. Demazeau, F. Dignum, J. Corchado, J. Bajo, R. Corchuelo, E. Corchado, F. Fernández-Riverola, V. Julián, P. Pawlewski y A. Campbell), vol. 71 de *Advances in Intelligent and Soft Computing*, páginas 493–500. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-12432-7.

SCHMIDT, D. C. Guest editor's introduction: Model-driven engineering. *Computer Journal*, vol. 39(2), páginas 25–31, 2006. ISSN 0018-9162.

SCHNEIDER, G., HOYMANN, C. y GOOSE, S. Adhoc personal ubiquitous multimedia services via UPnP. En *Proceedings of the 2nd IEEE International Conference on Multimedia and Expo*, vol. 1 de *ICME'01*, páginas 901–904. 2001. ISBN 0-7695-1198-8.

SERRANO, E. y BOTÍA, J. Validating ambient intelligence based ubiquitous computing systems by means of artificial societies. *Information Sciences*, vol. 222, páginas 3–24, 2013. ISSN 0020-0255.

SHIREHJINI, A. A. N. A generic UPnP architecture for ambient intelligence meeting rooms and a control point allowing for integrated 2d and 3d interaction. En *Proceedings of the 1st Joint Conference on Smart Objects and Ambient Intelligence*, vol. 1 de *sOc-EUSAI'05*, páginas 207–212. ACM, New York, NY, USA, 2005. ISBN 1-59593-304-2.

STEINBERG, D. y CHESHIRE, S. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005. ISBN 0596101007.

STEINER, D. FIPA: Foundation for Intelligent Physical Agents - das aktuelle schlagwort. *Künstliche Intelligenz*, vol. 12(3), página 38, 1998.

STRANG, T. y LINNHOFF-POPIEN, C. A context modeling survey. En *Proceedings of the 6th International Conference on Ubiquitous Computing - Workshop on Advanced Context Modelling Reasoning*, vol. 1 de *UbiComp'04*, páginas 1–8. 2004.

TAPIA, D., ABRAHAM, A., CORCHADO, J. y ALONSO, R. Agents and ambient intelligence: case studies. *Journal of Ambient Intelligence and Humanized Computing*, vol. 1(2), páginas 85–93, 2010. ISSN 1868-5137.

TAUBMAN, D. S. y MARCELLIN, M. W. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 2001. ISBN 079237519X.

TEAM, C. Caronte project. `http://innovation.logica.com.es/web/caronte/inicio`, 2011.

VENTURINI, V., CARBÓ, J. y MOLINA, J. M. Methodological design and comparative evaluation of a MAS providing AmI. *Expert Systems with Applications*, vol. 39(12), páginas 10656–10673, 2012. ISSN 0957-4174.

VENTURINI, V., CARBO, J. y MOLINA, J. M. Calor: Context-aware and location reputation model in ami environments. *Journal of Ambient Intelligence and Smart Environments*, vol. 5, páginas 589–604, 2013.

VINCENT, R., HORLING, B. y LESSER, V. An agent infrastructure to build and evaluate multi-agent systems: The java agent framework and multi-agent system simulator. En *Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems* (editado por T. Wagner y O. Rana), vol. 1887 de *Lecture Notes in Computer Science*, páginas 102–127. Springer Berlin Heidelberg, 2001. ISBN 978-3-540-42315-7.

VIZCAÍNO, A., GARCÍA, F., CABALLERO, I., VILLAR, J. y PIATTINI, M. Towards an ontology for global software development. *IET Software*, vol. 6(3), páginas 214–225, 2012. ISSN 1751-8806.

WEISER, M. Some computer science issues in ubiquitous computing. *Communications of the ACM*, vol. 36(7), páginas 75–84, 1993. ISSN 0001-0782.

WEISS, G. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1st edición, 2000. ISBN 0262731312.

WEST, M. y HARRISON, J. *Bayesian forecasting and dynamic models*. Springer-Verlag New York, Inc., New York, NY, USA, 2nd edición, 1997. ISBN 0-387-94725-6.

WINOGRAD, T. Architectures for context. *Human-Computer Interaction*, vol. 16(2), páginas 401–419, 2001. ISSN 0737-0024.

YORK, J. y PENDHARKAR, P. C. Human-computer interaction issues for mobile computing in a variable work context. *International Journal of Human-Computer Studies*, vol. 60(5–6), páginas 771–797, 2004. ISSN 1071-5819.

Lista de acrónimos

AmI	*Ambient Intelligence*, Inteligencia Ambiental
ANTS	*A Non sTop workerS*
API	*Application Programming Interface*
BDI	*Belief-Desire-Intention*
BPEL	*Business Process Execution Language*
CAKE	*Collaborative Agent-based Knowledge Engine*
CAMEO	*Context-Aware MiddlEware for Opportunistic mobile social networks*
CAWE	*Context-Aware Workflow Execution*
CoBrA	*Context Broker Architecture*
CoBrA-ONT	*CoBrA Ontology*
COCOMO	*COnstructive COst MOdel*
CORBA	*Common Object Request Broker Architecture*
CSC	Componente Sensible al Contexto, *Context-Aware Component*
DAML+OIL	Lenguaje de Marcado de Agentes de DARPA + Lenguaje de Intercambio de Ontologías, *DARPA Agent Markup Language + Ontology Interchange Language*
DAS	*Device Access Specification*
DNS	*Domain Name System*
FAERIE	*Framework for Ambient intelligence: Extensible Resources for Intelligent Environments*

FIPA-ACL .... *Foundation for Intelligent Physical Agents - Agent Communication Language*

FPU ..... Formación de Profesorado Universitario

GPS ..... Sistema de Posicionamiento Global, *Global Positioning System*

GRASIA ..... GRupo de investigación en Agentes Software: Ingeniería y Aplicaciones

GSM ..... Sistema Global para las comunicaciones Móviles, *Global System for Mobile communications*

HTTP ..... *HyperText Transfer Protocol*

IAM ..... Inteligencia Ambiental

ID ..... IDentificador

IDE ..... *Integrated Development Environment*, Entorno de Desarrollo Integrado

INGENIAS ... INGENIería para Agentes Software

INGENME ... *INGENIAS Meta-Editor*

JADE ..... *Java Agent DEvelopment Framework*

JAF ..... *Java Agent Framework*

JCAF ..... *Java Context-Aware Framework*

LDC ..... Líneas De Código

MASON ..... *Multi-Agent Simulator Of Neighborhoods*

OCP ..... *Open Context Platform*

ORM ..... *Object-Relational Model*

OSGI ..... *Open Services Gateway Initiative*

OWL ..... *Web Ontology Language*

P2P ..... *Peer to Peer*

R-OSGI ..... *Remote OSGi*

RMI ..... *Remote Method Invocation*

SMA ..... Sistema Multi-Agente

SOCAM ..... *Service-Oriented Context-Aware Middleware*

SociAAL ..... *Social Ambient-Assisted Living*

SSC ..... Sistema Sensible al Contexto

SUT ..... *System Under Test*, Sistema En Prueba

UCM ..... Universidad Complutense de Madrid

UDDI ..... *Universal Description, Discovery and Integration*

UM ..... Universidad de Murcia

UPnP ..... *Universal Plug and Play*

URI ..... *Uniform Resource Identifier*

URL ..... *Uniform Resource Locator*

—Fry, no hay nada más escrito, ¡sólo has hecho dos páginas de diálogos!
—Bueno, me llevó una hora escribirlas, pensé que darían para una hora.

Leela y Fry. Futurama.

El pasado es historia, el futuro un misterio,
el hoy es un regalo, por eso se le llama presente.

V de Vendetta.